Issue 32 | Sep-Oct 2017

DNCMagazine

www.dotnetcurry.com

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

What's upcoming in Xamarin.Forms 3

Continuous Testing in .NET

The State of Entity Framework Core

Resource management in complex C# applications

ASP.NET Core Web API Attributes

What's New for .NET Developers

Angular Universal Apps using Node.js

FROM THE EDITOR

Hello Readers,

I would like to start by thanking you for the fantastic response to our 5th Anniversary edition. We received some valuable feedback, over 120K downloads and some heartfelt appreciation. We have noted all your requests and suggestions and some of them will be implemented in the current as well as upcoming editions. We will also be including more of Azure and Machine Learning. So thank you all for the feedback and for making it worthwhile!

For this 32nd edition, we have a bouquet of exclusive tutorials for you covering .NET Core, Xamarin, ASP.NET Core, Angular, Design Practices, Entity Framework and Testing. We welcome our new authors and Microsoft MVPs, David Pine and Ricardo Peres.

Make sure to reach out to me directly with your comments and feedback on twitter via our handle @dotnetcurry or email me at suprotimagarwal@dotnetcurry.com.

I am looking forward to your feedback!

Editor in Chief

Suprotim Agarwal

suprotimagarwal@dotnetcurry.com

THE TEAM

Editor In Chief
Suprotim Agarwal

Art Director
Minal Agarwal

Contributing Authors
Damir Arh
David Pine
Gerald Versluis
Francesco Abbruzzese
Ravi Kiran
Ricardo Peres
Yacoub Massad

Technical Reviewers
Damir Arh
Daniel Jimenez Garcia
Mahesh Sabnis
Suprotim Agarwal
Yacoub Massad

Next Edition
November 2017

www.dotnetcurry.com/magazine

THANK YOU

FOR THE 32nd EDITION

@yacoubmassad

@jfversluis

@F_Abbruzzese

@damirarh

@davidpine7

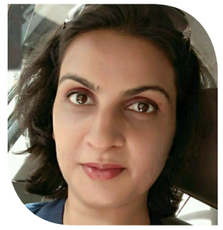
@RJPeres75

@maheshdotnet

@sravi_kiran

@suprotimagarwal

@dani_djg



@ saffronstroke

WRITE FOR US

mailto: suprotimagarwal@dotnetcurry.com

Why **Fortune 500** companies choose **RavenDB**?

In a world where data is one of the most important assets of any business the database technology should not only be protecting its data but also enhancing its business.

To address both of those needs, Hibernating Rhinos has introduced its NoSQL database called RavenDB and for the past few years, due to enhanced capabilities, it has become the choice of Fortune 500 companies.

The protection of data comes with meeting all the ACID parameters, being fully transactional and having extended failover support to guarantee you that the data will be safe and sound even when node failure happens. Moreover, the extended replication features allow businesses to setup complex failover clusters to move their protection to the next level and ensure availability or enhance their work by enabling sophisticated sharding and load balancing capabilities.

The out of the box querying features, high-performance and self-optimization assure that the database will not stand in the way of company growth.

All this is provided with user-friendly HTML5 management interface, ease of deployment and top-notch C# and Java client libraries.

- Schema-free
- Scalable
- RavenFS
- Easy to use
- Transactional
- High Performance
- Extensible
- Designed with Care

NEW

- Monitoring
- Hot Spare
- Clustering

RAVENDB 3.5 RELEASED

ravendb.net

CONTENTS

Gerald Versluis

Xamarin.Forms 3 What's upcoming?

What is Xamarin.Forms?

Before we dive right in and show you what to expect from the latest Xamarin.Forms release, let me first tell you a little bit about what Xamarin.Forms is exactly.

If you have worked with, or looked up Xamarin before, you'll probably know by now that it is a great solution to create multi-platform apps based on C# and .NET. You can use it to share all of your business logic throughout all of your apps across Android, iOS, UWP and even across a lot more platforms these days like the Raspberry Pi, Apple TV, etc.

While this is great in itself, Xamarin took it a step further and added the missing link.

At the Microsoft Build conference this year, Xamarin presented the next iteration of Xamarin.Forms v3. In this article, I will show you a few cool features that they announced and what you can expect from this new version.

With 'traditional Xamarin', as they now call it, you can do everything I have mentioned before, but you will still have to put in efforts in the user interface (UI) for each platform.

And that is just fine!

You can leverage the powerful tools that are already available for the respective platform to create a UI that integrates seemingly with the OS that you are targeting. On May 28th of 2014, Xamarin introduced Xamarin.Forms as part of Xamarin version 3.

With Forms, Xamarin then enabled developers to also share the UI code across all platforms.

An abstraction layer was introduced with which an Entry or Label could be defined. Then, with the power of Xamarin.Forms, it would be translated to the native equivalent on the platform that the app is running on at that time.

For instance, this would be a UITextView for iOS, a EditText for Android and a TextBox on Windows in the case of an Entry. To define this shared UI code, you can either use XAML – a XML-based language to define user interfaces – but you can also define the UI in C# code.

In the fall of 2017, Xamarin Forms is coming up with its next milestone version : Xamarin.Forms 3.

Note: *The versioning of Xamarin and Xamarin.Forms are not aligned. Ahead of this release, I will walk you through some of the key features that they have already announced, and are even available for you today!*

Coming to you in Xamarin.Forms 3

Besides a whole lot of bugfixes and improvements, these are some cool features that you want to look for in the new version of Forms.

Xamarin.Forms Embedding

Since half-way 2016, there is already a thing called 'native embedding' in Xamarin.Forms.

With this feature, you can embed native controls that do not have a Forms abstract counterpart (yet). Some good examples of this are the UISegmentedControl on iOS and the Floating Action Button (FAB) on Android.

These controls are so specific and characteristic to their respective platforms, that they are unlikely to get an abstraction, so you can use it across platforms. With native embedding, the Xamarin team has done a great job in finding a way to add these controls to that specific platform, but in a generic way.

You can read more on this here: https://blog.xamarin.com/embedding-native-controls-into-xamarin-forms/.

Xamarin.Forms Embedding, is just the other way around.

With this method, you can embed Xamarin.Forms controls, pages and all other elements into your traditional Xamarin app, with traditional Xamarin being an app written in C# but using the native UI elements. This enables you to gradually transition from your old Xamarin app, into a new one and with an even more code-sharing Xamarin.Forms app.

Not really a part of the new Forms version, but still nice to mention in this context is the Embeddinator-4000. This application allows you to convert a .NET library into an Objective-C and/or Java one. This way, you can even start converting your existing native apps to Xamarin in phases.

More platforms are supported

Xamarin already enables you to develop C# code for a wide variety of platforms. Most obvious are the phone and tablet platforms: Android, iOS and Windows.

But did you know you could also run your code on a PlayStation 4, Xbox, Google Glass, Amazon Kindle, Linux, etc?

Here yet again, watch out that it does not mean that whenever Xamarin runs somewhere, Xamarin.Forms is supported as well. The platforms that are supported in Forms would be: Android, iOS, Windows Phone and UWP. Most recently Mac OS was added, the desktop OS by Apple.

And now, support for some new platforms are coming up!

The team is working hard to support GTK#, a .NET wrapper for GTK+, the UI toolkit mainly used on Linux system. This means that you can also easily design forms for a wide variety of Linux distributions in the near future. And WPF will be a supported platform as well, I will talk about that a little bit later on in the XAML Standard section.

Enhancements and more speed

The new major release of Forms will mostly be about stability and improvement. This means that they have gone back to the drawing board with the current version and took a long and good look on where improvements could be made.

Control renderers

When we take a closer look at the inner workings of Xamarin.Forms, we will find an important role furnished in the so-called 'renderers'.

It is the responsibility of the Forms libraries to translate – or better yet, render the elements that you define in an abstract manner, to their native counterpart. Have a look at the figure below to see how that works.

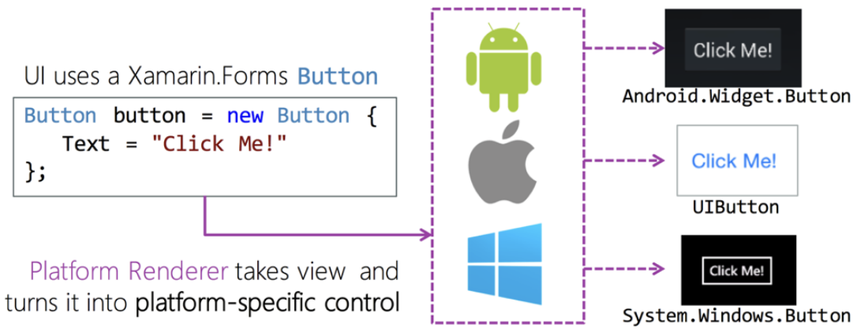


Figure 1: Overview of how a renderer does its work. Image courtesy by Xamarin.

Because this is a large part of the Xamarin.Forms solution, you can imagine that optimizing this process can make a tremendous difference. The next generation of renderers that they are implementing now are called *fast renderers*.

Layout compression

The renderers discussed earlier, mostly do their work at runtime. Whenever a page is requested, it will take that page and go through the whole visual tree and start rendering all the controls that are on it. This is time consuming and also reflects on the memory usage.

In the version of Forms that is to be released, layout compression will be available. When enabled, the layouts will be optimized at compile time and thus improve performance at run time.

New binding type

One of the Xamarin Forms pillars, is data-binding.

This supports the use of architectural patterns like the MVVM pattern. In short, you do not need to reference actual controls by their name to give them a specific value. Instead, you turn it around; you tell the control from which property it needs to take the value, in your code-behind.

This way, your code is more decoupled because you do not have to refactor your code when a new UI is developed. Just make sure the new UI also references the right properties in your code and you are good to go.

Data-binding can have a great impact on your performance. There are a few different data-binding modes.

You can supply the value to the control, which is one-way. But you could also require the control to update the value in your code, so the value goes both ways and is therefore known as two-way and there some other, less significant ones.

Every time a value needs to be updated, depending on the mode, a few cycles through your code are fired off. This means, when you have a page with a number of data-bound controls, performance could go down.

To make it more performant, Xamarin is now introducing a new data-binding mode: **one-time binding**.

Note: *Do not confuse this with the one-way binding mode. The one-time binding mode, as you could imagine from the name, just ingests the value of the property it is bound to once, the first time the binding is being set up. After that, the binding is not being evaluated anymore.*

More flexibility

If you have some affinity with web development, you might know (and love) the FlexLayout (or Flexbox, more information: https://css-tricks.com/snippets/css/a-guide-to-flexbox/) system. This system is being translated into a Xamarin.Forms version and incorporated in this new version they are releasing.

For those who aren't very familiar with web development; the Flexbox layout system provides for a way to efficiently lay out, distribute and align space amongst items in a container. This also works for elements

with an unknown size, hence the word 'flex'.

There is not much known right now on how Xamarin wants to implement this, but looking at the Flex system in web applications, we might see more out-of-the-box responsiveness and automatic calculations of UI elements.

Improved styling options

Together with the FlexLayout system, they are also expanding the Forms styling options.

And again, if you are a web developer in disguise, you will love this! CSS has proven to be an easy-to-learn, yet powerful way of styling user interfaces. And now, the team at Xamarin is actually implementing more CSS-like styling options into Xamarin.Forms.

Although there is no actual detail available yet, it is a much-heard request from the community. And as a big fan of everything that is simple, but effective, I can't wait to see what they have in store for us with this.

XAML Standard

One of the first critical notes that was heard after the release of Xamarin.Forms, was the incompatibility with the WPF XAML.

Although most of it has the same syntax, the XAML definition used in Forms deviates in some areas from the syntax used in WPF. For example, in WPF there is a layout element which is called a `StackPanel`. In Xamarin they named it a `StackLayout`.

Initially, I presume, this was done for two reasons. One, the naming of the WPF components did not always sound logical for the use on mobile. Secondly, the team at Xamarin wanted to prevent developers from just copy and pasting their (desktop) WPF layout to a Xamarin app without rethinking the layout.

If developers would be able to copy layouts from their existing apps, there probably would have been a lot of unusable apps out there right now. Because the design paradigms for desktop applications just aren't suitable for mobile.

To overcome the confusion of the different XAML dialects, Microsoft has announced **XAML Standard**.

With XAML Standard, a unified language will be developed and you *will* be able to exchange layouts between Windows 10 and Xamarin Forms and whichever platforms that will support XAML Standard in the future.

Besides the way you define your controls, nothing will change. XAML will still be the way to define your UI in an abstract matter and the controls will still be rendered to their native counterpart.

Looking at the draft of XAML Standard version 1, which can be found here: https://github.com/Microsoft/xaml-standard/blob/staging/docs/v1draft.md, it looks like the naming will go back to the WPF/Windows 10 dialect. As a Forms developer, that will take some getting used to.

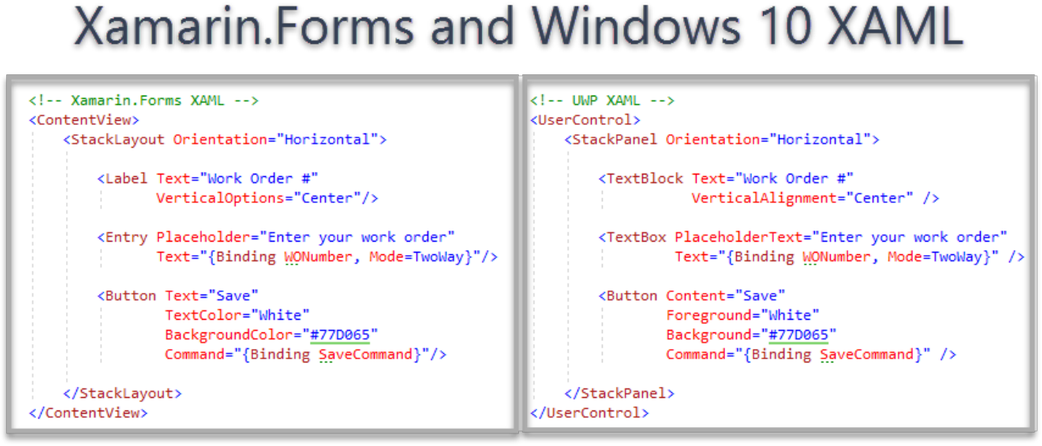


Figure 2: A sample of XAML Standard based on the first draft. Image courtesy by Microsoft.

Work with these hot bits, today!

If you do not want to wait for the official release, you can already take a look at some parts, right now. If you want to join in, or have a look at the discussion on XAML Standard, you can visit this link: https://aka.ms/xamlstandard.

Or if you want to have a look at the Xamarin.Forms Embedding, you can follow these steps:

- Add the proprietary Xamarin.Forms feed to your IDE: https://www.myget.org/F/xamarinforms-dev/api/v3/index.json
- Clone the demo application created by David Ortinau: https://github.com/davidortinau/build2017-new-in-xamarin-forms

You should be able to restore the NuGet package needed, Xamarin.Forms 3.0.0.100-embeddingpreview, to the project.

The rest of the features are still being worked on internally and are not yet out. But of course, Forms is open-source these days, so you should be able to see some of the work on the repository: https://github.com/xamarin/Xamarin.Forms/

Final thoughts

This is just a handful of features and improvements that are coming to Xamarin.Forms.

To read up on everything that is planned, take a look at the roadmap, which is on the Forums, here: https://forums.xamarin.com/discussion/85747/xamarin-forms-feature-roadmap/p1.

Most notable is that 'only' two new controls are planned and the rest is targeting performance and stability

mainly. But still, things like packaging Forms into one single DLL, implementing a Visual State Manager, supporting .NET Standard 2.0, adding G18n support and much more are on the list as well. There will be enough new awesome stuff to work with.

All these goodies are planned for the third quarter of 2017, which has probably arrived by the time you are reading this. But of course some of the features could have been pushed back and might be available to use later.

I hope that you are as enthusiastic as I am about this next step in the evolution of Xamarin.Forms, and will create even more awesome apps with these goodies. If you have any questions or comments, or you want to show off your Xamarin app, please do not hesitate to contact me

Microsoft MVP Most Valuable Professional

Gerald Versluis
Author

Gerald Versluis (@jfversluis) is a full-stack software developer and Microsoft MVP (Xamarin) from Holland. After years of experience working with Xamarin and .NET technologies, he has been involved ina number of different projects and has been building several apps. Not only does he like to code, but he is also passionate about spreading his knowledge - as well as gaining some in the bargain. Gerald involves himself in speaking, providing training sessions and writing blogs (https://blog.verslu.is) or articles in his free time. Twitter: @jfversluis Email: gerald@verslu.is Website: https://gerald.verslu.is

Thanks to Yacoub Massad for reviewing this article.

Francesco Abbruzzese

Building Web Applications with **Knockout.js** and **ASP.NET core**

Amongst all the client side frameworks backed by big companies, React.js and Angular.js appear to be the most popular. However, Knockout- 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.

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.

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 that assists developers with client side technologies and can execute JavaScript Node.js code on the server side. Among them, Microsoft.AspNetCore.SpaTemplates installs spa templates for the more common client side frameworks: Angular 2, React.js, Knockout.js, and Aurelia.

All templates use webpack 2 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:

```
Templates                                         Short Name      Language      Tags
--------------------------------------------------------------------------------------------
Console Application                               console         [C#], F#      Common/Console
Class library                                     classlib        [C#], F#      Common/Library
Unit Test Project                                 mstest          [C#], F#      Test/MSTest
xUnit Test Project                                xunit           [C#], F#      Test/xUnit
ASP.NET Core Empty                                web             [C#]          Web/Empty
ASP.NET Core Web App                              mvc             [C#], F#      Web/MVC
MVC ASP.NET Core with Angular                     angular         [C#]          Web/MVC/SPA
MVC ASP.NET Core with Aurelia                     aurelia         [C#]          Web/MVC/SPA
MVC ASP.NET Core with Knockout.js                 knockout        [C#]          Web/MVC/SPA
MVC ASP.NET Core with React.js                    react           [C#]          Web/MVC/SPA
MVC ASP.NET Core with React.js and Redux          reactredux      [C#]          Web/MVC/SPA
MVC ASP.NET Core with Vue.js                      vue             [C#]          Web/MVC/SPA
ASP.NET Core Web API                              webapi          [C#]          Web/WebAPI
Solution File                                     sln                           Solution

Examples:
    dotnet new mvc --auth None --framework netcoreapp1.1
    dotnet new angular --Framework netcoreapp1.1
    dotnet new --help
```

Here choose the template whose short name is "knockout".

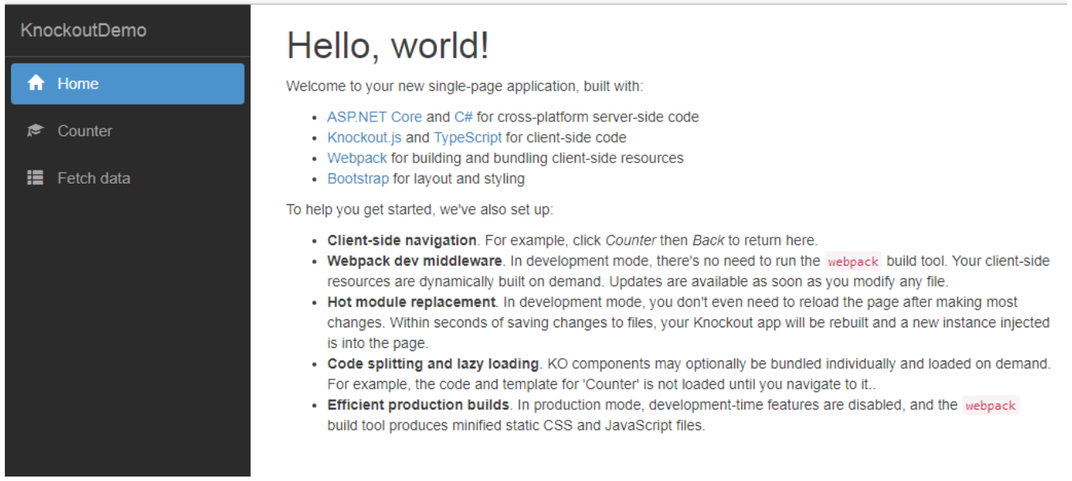
Create a folder called "KnockoutDemo" for our project. Then open a console in this directory. In Windows, we can do this by holding down "shift" while right clicking on the newly created folder, and then selecting "open command window here".

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 by the code in wwwroot\ClientApp\router.ts, that in turn uses Crossroad.js to handle routes (url-components mappings). After that, knockout.js components are retrieved by the custom knockout.js component loader 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

.

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 bundles all files is defined into the webpack.config.js and webpack.config.vendor.js configuration files.

webpack.config.vendor.js uses the DllPlugin to bundle all needed npm modules as a library that is then referenced by webpack.config.js with the DllReferencePlugin 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 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!’ 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 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 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 npm package that is a window. fetch polyfill that works both server side with node.js, as well as on client side. More specifically, it uses whatwg-fetch on client side and 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>
```

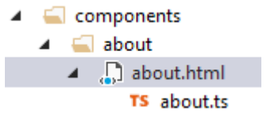
While data is being downloaded from the server, the forecasts property is empty, so the "ifnot" binding shows the paragraph content. Finally, when "forecast" observable is filled with data, the "foreach" binding shows the table rows bound to each item properties through the text binding.

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.

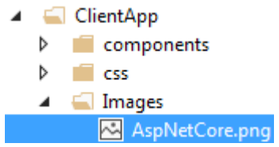
Adding an image to the new page

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 also 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.



Add an "Images" folder to the "ClentApp" folder and add the "AspNetCore.png" you may 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')
};
```

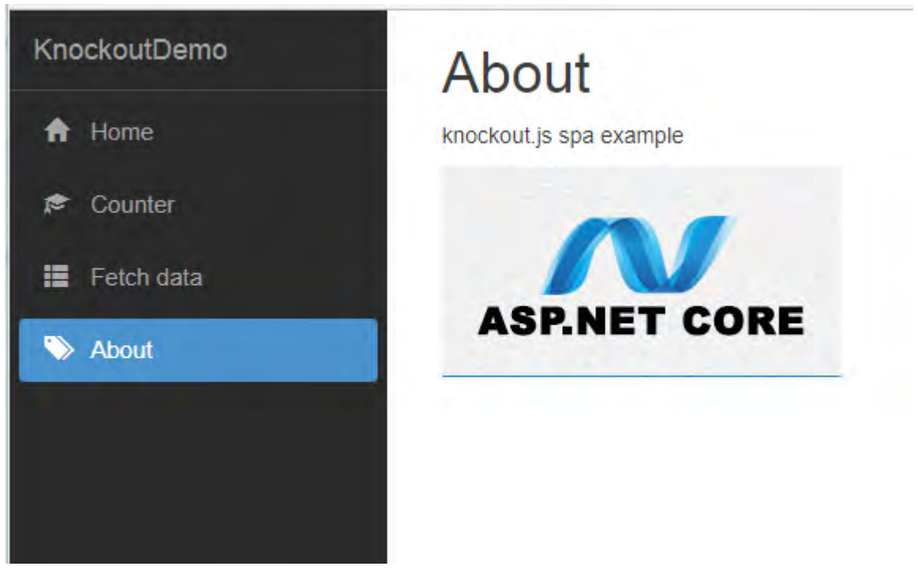
We used "require" since with an "import" statement, the TypeScript compiler would have signalled an error because TypeScript import statement works only with TypeScript and JavaScript files.

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 a 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:

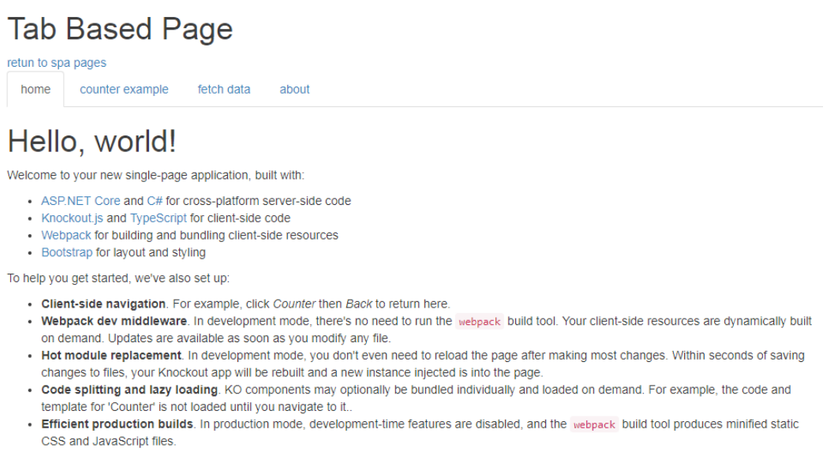
```
entry: {
  'main': './ClientApp/boot.ts',
  'tab': './ClientApp/tab.ts'
}
```

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 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 from GitHub at
bit.ly/dncm32-knockout-aspcore

Microsoft® Most Valuable Professional

Francesco Abbruzzese
Author

Francesco Abbruzzese (@F_Abbruzzese) implements ASP.NET MVC applications, and offers consultancy services since the beginning of this technology. He is the author of the famous Mvc Controls Toolkit, and his company (www.mvc-controls.com/) offers tools, and services for ASP.NET MVC. He moved from decision support systems for banks and financial institutions, to the Video Games arena, and finally started his .NET adventure with the first .NET release. He also writes about .NET technologies in his blog: www.dotnet-programming.com/

Thanks to Daniel Jimenez Garcia for reviewing this article.

Damir Arh

Continuous Testing in .NET

What is Continuous Testing?

A very important factor in effective unit testing is how long it takes for the developer to see the test results.

The shorter the time, the more beneficial are the tests. If the tests are only run as part of the nightly build, information about any of them failing will only be available the next morning. By then, the developer would have most likely forgotten some details about the changes that were made the previous day, especially if in the meantime, he/she started working on something else.

Building the project, running the tests on the server immediately after code changes and a commit to source control, mostly avoids this problem, as the results are delayed only for the duration of the build.

Of course, the developer can also run the tests by himself, locally in his own development environment before committing the code. This can save some additional time and prevent him from committing broken code in the first place. But even in this case, he must consciously run the tests in order to see the results, which he will typically do when he has completed some piece of work.

The idea of continuous testing is to make this feedback loop even more tight.

The tests should be run automatically whenever the code changes and the developer should not need to run them manually. This can prove useful even if the developer only writes tests, after he has written the code under test. When he needs to refactor or enhance existing code, these tests will take on the role of regression tests, i.e. they will serve as a safety net from breaking original functionality.

However, continuous testing shows its real worth when the tests are written in advance or in parallel with the code under test, as required by Test Driven Development (TDD).

With the tests being run all the time while the developer is writing the code, test results provide an up-to-date view into the current state of the code – which parts are already implemented correctly and which are still missing or currently broken. This makes the development more effective and satisfying as any fix to the code, results in an almost immediate change in test results.

Continuous Testing Support in Development Tools

Continuous testing is not possible if the development tools don't offer support for it.

To my knowledge, the first tool to allow continuous testing in the .NET ecosystem was Mighty Moose, also known as ContinuousTests – a Visual Studio extension and command line runner, which started out as a commercial tool, but was later made free and open sourced. Now, it seems completely abandoned and outdated.

This first attempt was soon followed by other third party commercial extensions for Visual Studio. Today, there are three competing solutions available:

- NCrunch from Remco Software,
- DotCover by JetBrains, sold as a part of ReSharper Ultimate bundle, and
- Smart Runner by Typemock, sold together with Typemock Isolator for .NET.

Microsoft's first shy attempt in this field was a feature that was added to Test Explorer: *Run Tests After Build switch* made it possible to automatically run the tests after every build.

While this feature didn't really allow for real continuous testing, it was nevertheless the first step in that direction. A full solution for continuous testing was introduced in Visual Studio 2017 as a feature named Live Unit Testing.

At about the same time, a similar feature was added to the command line tools for .NET Core. You can use the dotnet-watch file watcher to monitor the source code for changes and run the tests in response. It does not integrate with Visual Studio and reports results only to the console window, but can therefore be used with any code editor and on any platform.

In the remainder of this article, we will take a closer look at Microsoft's current solutions for continuous testing.

Live Unit Testing

Live Unit Testing functionality is only available in the Enterprise edition of Visual Studio 2017.

In the final release of Visual Studio 2017, which was released in March, support was limited to projects targeting the full .NET framework. If you want to use it with .NET Core (1.0, 1.1 or the preview of 2.0), you will also need to have update 15.3 installed. At the time of writing, it was only available as a preview, which could be safely installed side-by-side with an existing release version of Visual Studio 2017 without interfering with it.

You can use any of the three most popular test frameworks (MSTest, NUnit and xUnit.net) with Live Unit Testing, however you need to use a recent enough version for all of them, e.g. there's no support for NUnit 2 and MSTest v1.

Setting up an ASP.NET Core Project

To give Live Unit Testing a try, we will create a new project, based on the *ASP.NET Core Web Application (.NET Core) project template. We will choose the Web Application option with No Authentication and Enable Docker Support.*

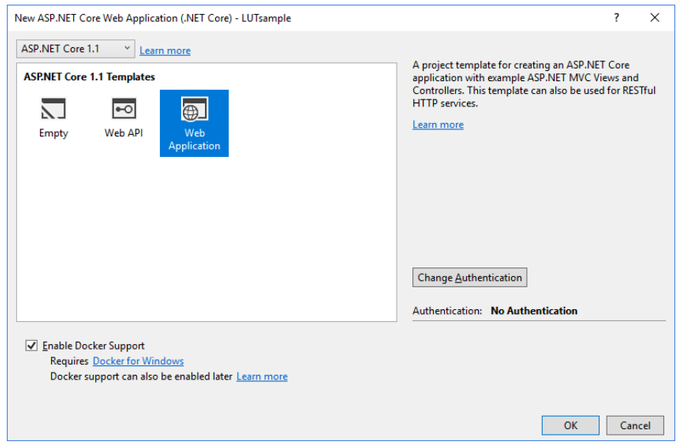


Image 1: New ASP.NET Core project

With these settings, Visual Studio will automatically run the application inside a Linux container on your Windows machine. Of course, this requires Docker for Windows to be installed.

For the tests, we need to add another project to our solution.

As we want to use MSTest test framework, we will choose the *Unit Test Project (.NET Core)* project template. For xUnit.net tests, we would need to choose the *xUnit Test Project (.NET Core)* project template instead.

Both project templates take care of installing the test framework and test adapter Nuget packages for the selected test framework, which are required for Visual Studio to recognize the tests and run them. We should also add a reference to the web project from the test project, so that we can later access the application classes from the tests.

To check that everything was set up correctly, we can now *Run All* the tests from Test Explorer. It should detect the empty test in the newly created test project and run it. Although the application project is configured to run in Docker, this does not affect the test project. Visual Studio will still run the tests locally in its own dedicated sandbox.

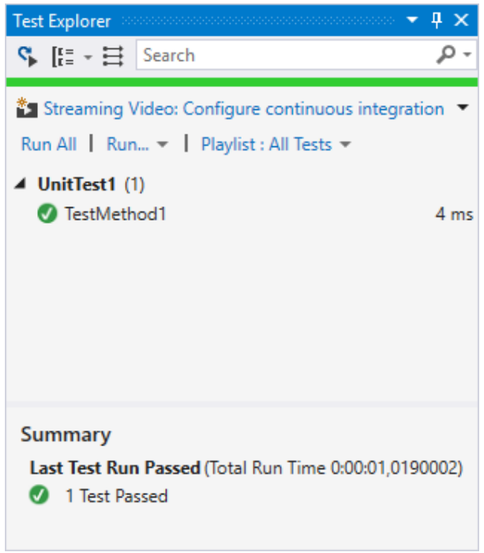


Image 2: Run All tests in Test Explorer

It's now time to enable Live Unit Testing for our solution via the Start command in the *Test > Live Unit Testing menu*. To disable it again at a later time, we could use the Pause or Stop command from the same menu.

It's also worth mentioning that by default you will need to reinitialize Live Unit Testing every time you restart Visual Studio. You can change this behavior with a setting on the *Live Unit Testing page* of the *Options dialog: Start Live Unit testing on solution load.*

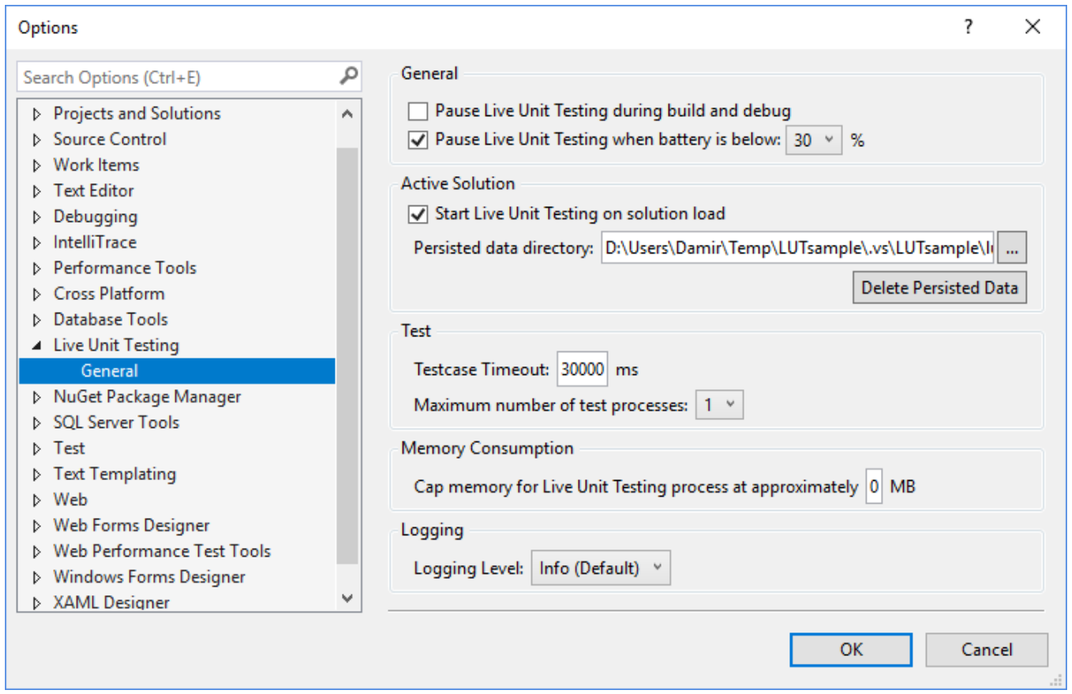


Image 3: Start Live Unit testing on solution load

A Typical Development Cycle

Live Unit Testing shines best when we adhere to Test Driven Development practices and write tests at the same time we are writing the code under test.

Let's start by adding a service class to our ASP.NET Core project:

```
using System;
namespace LUTsample.Services
{
  public class FibonacciService
  {
    public int Calculate(int n)
    {
      throw new NotImplementedException();
    }
  }
}
```

The `Calculate` function will eventually return the n-th number from the Fibonacci sequence. Until we implement it, it seems appropriate that it throws a `NotImplementedException`. Before we start implementing it, we should first write a test, specifying the expected result:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using LUTsample.Services;

namespace LUTsample.Tests
{
  [TestClass]
  public class FibonacciTest
  {
    [TestMethod]
    public void Calculate1()
    {
      var fibonacci = new FibonacciService();
      Assert.AreEqual(1, fibonacci.Calculate(1));
    }
  }
}
```

The test of course fails.

This is also clearly indicated in the code editor window by Live Unit testing as soon as we write the test.

```
1   using Microsoft.VisualStudio.TestTools.UnitTesting;
2   using LUTsample.Services;
3
4   namespace LUTsample.Tests
5   {
6       [TestClass]
        0 references
7       public class FibonacciTest
8       {
9           [TestMethod]
            0 references | 0 exceptions
10          public void Calculate1()
11          {
12              var fibonacci = new FibonacciService();
13              Assert.AreEqual(1, fibonacci.Calculate(1));
14          }
15      }
16  }
```

Image 4: Failing test

The glyphs in front of each line of code are an effective way of showing information about test results and code coverage:

- A **green check** indicates that the line is executed by at least one test and that all covering tests pass.
- A **red X** indicates that the line is executed by at least one test and that at least one of those tests fails.
- A **blue dash** indicates that the line is not executed by any test.

```
        2 references | 1/2 passing | 0 exceptions
5       public int Calculate(int n)
6       {
7           var fibonacci = new int[n + 1];
8           fibonacci[0] = 0;
9           fibonacci[1] = 1;
10
11          for (int i = 2; i <= n; i++)
12          {
13              fibonacci[i] = fibonacci[i - 1] + f
14          }
15
16          return fibonacci[n];
17      }
18  }
```

Image 5: Different Live Unit Testing glyphs

It's time to make the test pass by implementing the `Calculate` function:

```
public int Calculate(int n)
{
  var fibonacci = new int[n + 1];
  bonacci[0] = 0;
  fibonacci[1] = 1;

  for (int i = 2; i <= n; i++)
  {
    fibonacci[i] = fibonacci[i - 1] + fibonacci[i - 2];
  }

  return fibonacci[n];
}
```

Not only does the test pass now, but the coverage information is also shown in the code window.

```
1  namespace LUTsample.Services
2  {
       2 references
3      public class FibonacciService
4      {
           1 reference | 1/1 passing | 0 exceptions
5          public int Calculate(int n)
6          {
7              var fibonacci = new int[n + 1];
8              fibonacci[0] = 0;
9              fibonacci[1] = 1;
10
11             for (int i = 2; i ⩽ n; i++)
12             {
13                 fibonacci[i] = fibonacci[i - 1] + fibonacci[i - 2];
14             }
15
16             return fibonacci[n];
17         }
18     }
19 }
20
```

Image 6: Coverage information in code editor window

With one line of code not being executed by our test, we really need to add more tests:

```
[TestMethod]
public void CalculateOutOfRange()
{
  var fibonacci = new FibonacciService();
  Assert.ThrowsException<ArgumentOutOfRangeException>(
  () => fibonacci.Calculate(-1));
}

[TestMethod]
public void Calculate1()
{
  var fibonacci = new FibonacciService();
  Assert.AreEqual(1, fibonacci.Calculate(1));
}

[TestMethod]
public void Calculate8()
{
  var fibonacci = new FibonacciService();
  Assert.AreEqual(21, fibonacci.Calculate(8));
}
```

All lines of code are now covered by tests. However, one of the tests fails:

```
1  namespace LUTsample.Services
2  {
       4 references
3      public class FibonacciService
4      {
           3 references | 2/3 passing | 0 exceptions
5          public int Calculate(int n)
6          {
7              var fibonacci = new int[n + 1];
8              fibonacci[0] = 0;
9              fibonacci[1] = 1;
10
11             for (int i = 2; i ⩽ n; i++)
12             {
13                 fibonacci[i] = fibonacci[i - 1] + fibonacci[i - 2];
14             }
15
16             return fibonacci[n];
17         }
18     }
19 }
20
```

Image 7: Coverage information with failing tests

If we click on any glyph, a pop-up window will list all of the covering tests for that line. If we hover over a failing test in this list, more information about the failure will be shown in the tooltip.

Double-clicking a test in the list will navigate us to the test.

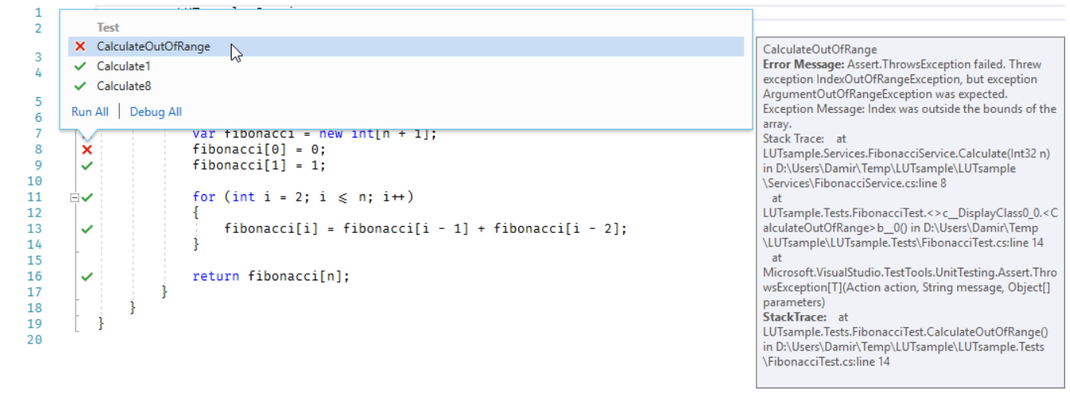


Image 8: Tooltip with information about a failing test

The next logical step would be fixing the function code so that all the tests will pass, but I'm leaving that to you as an exercise.

By now you should already have a pretty good idea of how the development cycle looks when practicing Test Driven Development with continuous testing. Once you get used to it, this approach should improve your code quality and make you more productive.

.NET Core Command Line Tools

.NET Core SDK includes a set of command line tools, which can be used as an alternative to the Visual Studio graphical user interface. We will use them to prepare a similar working environment with continuous testing.

Setting Up the Project

Let's start by creating a new solution with a default ASP.NET Core MVC web application project inside it:

```
md LUTsample
cd .\LUTsample\
dotnet new sln -n LUTsample
dotnet new mvc -n LUTsample -o LUTsample
dotnet sln add .\LUTsample\LUTsample.csproj
```

Just like the project template in Visual Studio, the `mvc` template creates a working web application. Unfortunately, there's no template available yet to setup deployment to a Docker container from command line, therefore this application will run locally. We only need to restore the NuGet packages and we're ready to go:

```
dotnet restore
cd .\LUTsample\
dotnet run
```

We can open the web page at http://localhost:5000 in our favorite browser, as explained in the output of the last command:

```
Hosting environment: Production
Content root path: D:\Users\Damir\Documents\LUTsample\LUTsample
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

It's time to create an accompanying test project and add it to the solution. Let's stop the local web server with Ctrl+C and execute the following commands:

```
cd..
dotnet new mstest -n LUTsample.Tests -o LUTsample.Tests
cd .\LUTsample.Tests\
```

The mstest template will create a new test project based on the MSTest v2 testing framework with a sample empty test. The xunit template will use xUnit.net instead. After restoring the NuGet packages we can run the tests:

```
dotnet restore
dotnet test
```

Of course, the empty test passes:

```
Build started, please wait...
Build completed.

Test run for D:\Users\Damir\Documents\LUTsample\LUTsample.Tests\bin\Debug\
netcoreapp1.1\LUTsample.Tests.dll(.NETCoreApp,Version=v1.1)
Microsoft (R) Test Execution Command Line Tool Version 15.0.0.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...
```



```
Total tests: 1. Passed: 1. Failed: 0. Skipped: 0.
Test Run Successful.
Test execution time: 0.9738 Seconds
```

Running the Tests Continuously

We need to start modifying the code now.

Since command line tools take care of all .NET Core specific operations, we could use any code editor. I chose Visual Studio Code for this article, which is available for Windows, Linux and macOS just like ASP.NET Core itself.

Editorial Note: *For a quick primer on VS Code, check http://www.dotnetcurry.com/visualstudio/1340/visual-studio-code-tutorial*

With the C# extension installed, as soon as we open the solution folder in the editor, Visual Studio Code will offer to automatically generate the build and launch definitions for our solution.

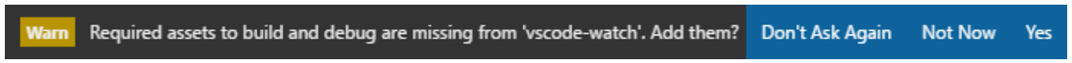


Image 9: Automatically create build and debug assets

This will create `launch.json` and `tasks.json` files in the `.vscode` subfolder, allowing us to quickly build the project with Ctrl+Shift+B and debug it with F5.

We're going to follow the same process as with Visual Studio 2017.

First we'll add the `FibonacciService` with the not yet implemented `Calculate` function to a new Services folder inside the `LUTsample` project folder. Next, we'll replace the empty test in the `LUTsample` test project with a real one.

For the test project to still build successfully, we need to add a reference to the web application:

```
dotnet add reference ..\LUTsample\LUTsample.csproj
```

We could now run this new failing test with `dotnet test`, but since we want to run the tests continuously, we need to add the dotnet-watch tool to the test project, following the instructions in its GitHub repository.

Open the LUTsample.Tests.csproj file in the editor and add the following inside its `Project` element:

```
<ItemGroup>
  <DotNetCliToolReference Include=”Microsoft.DotNet.Watcher.Tools” Version=”1.0.1”
/>
</ItemGroup>
```

This will work with .NET Core 1.0 and .NET Core 1.1. For .NET Core 2.0 projects we need to use version 2.0.0 of Microsoft.Dotnet.Watcher.Tools instead.

After restoring the new NuGet package, we can instruct the watch tool to run the test:

```
dotnet restore
dotnet watch test
```

It will output details about the failing test, but unlike `dotnet test`, it will keep running and watch for any changes in either the application or the test project:

```
watch : Started
Build started, please wait...
Build completed.

Test run for D:\Users\Damir\Temp\LUTsample\LUTsample.Tests\bin\Debug\netcoreapp1.1\
LUTsample.Tests.dll(.NETCoreApp,Version=v1.1)
Microsoft (R) Test Execution Command Line Tool Version 15.0.0.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...
Failed   LUTsample.Tests.FibonacciTest.Calculate1
Error Message:
 Test method LUTsample.Tests.FibonacciTest.Calculate1 threw exception:
System.NotImplementedException: The method or operation is not implemented.
Stack Trace:
    at LUTsample.Services.FibonacciService.Calculate(Int32 n) in D:\Users\Damir\
Temp\LUTsample\LUTsample\Services\FibonacciService.cs:line 9
   at LUTsample.Tests.FibonacciTest.Calculate1() in D:\Users\Damir\Temp\LUTsample\
LUTsample.Tests\FibonacciTest.cs:line 13


Total tests: 1. Passed: 0. Failed: 1. Skipped: 0.
Test Run Failed.
Test execution time: 0.8896 Seconds


watch : Exited with error code 1
watch : Waiting for a file to change before restarting dotnet...
```

As soon as we add the implementation for the `FibonacciService.Calculate` method and save the changes, the watch tool will rebuild the projects and rerun the tests:

```
watch : Started
Build started, please wait...
Build completed.

Test run for D:\Users\Damir\Temp\LUTsample\LUTsample.Tests\bin\Debug\netcoreapp1.1\
LUTsample.Tests.dll(.NETCoreApp,Version=v1.1)

Microsoft (R) Test Execution Command Line Tool Version 15.0.0.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...

Total tests: 1. Passed: 1. Failed: 0. Skipped: 0.
Test Run Successful.
Test execution time: 0.8763 Seconds

watch : Exited
watch : Waiting for a file to change before restarting dotnet...
```

Unlike Live Unit Testing, the results and the coverage are not shown directly in the code editor, but we still get immediate feedback with every change. If we now add the additional tests, they will again be immediately run, indicating that one of them fails.

For additional convenience you could even run dotnet-watch directly in the terminal that's built into Visual Studio Code. This way you don't need to have an extra terminal window open all the time.

Conclusion:

We have explored how continuous testing tools can make testing a more integral part of your development process, whether you're following TDD or not.

Support for continuous testing is becoming more common in the .NET ecosystem and will only improve in the future. The .NET Core command line tools can be used with any editor and on any platform. Visual Studio 2017 introduces the Live Unit Testing feature, which is only included in the Enterprise edition. For other editions there are alternative third-party commercial extensions available ■

Microsoft MVP Most Valuable Professional

Damir Arh
Author

Damir Arh has many years of experience with Microsoft development tools; both in complex enterprise software projects and modern cross-platform mobile applications. In his drive towards better development processes, he is a proponent of test driven development, continuous integration and continuous deployment. He shares his knowledge by speaking at local user groups and conferences, blogging, and answering questions on Stack Overflow. He is an awarded Microsoft MVP for .NET since 2012.

Thanks to Yacoub Massad for reviewing this article.

Yacoub Massad

This article discusses problems and solutions related to resource management in complex C# applications.

Resource management in complex C# applications

Introduction

When writing software applications, many a time we need to deal with resources such as files, database connections, Windows Communication Foundation (WCF) proxies, etc.

What is essential about such resources, is that we need to manage them *correctly*.

For example, when a file is **opened**, read or written to, it should be **closed**. We need to make sure that we always close the file even if an error occurred during processing of data.

Consider the following example:

```
private static void ProcessFile()
{
  var fileReader = new StreamReader(“c:\\file.txt”);
  while (true)
  {
    var line = fileReader.ReadLine();
    if (line == null)
      break;
      ProcessLine(line);
  }
  fileReader.Close();
}
```

This code opens file.txt by invoking the constructor of the `StreamReader` class. It then loops to read and process the lines of the file, one by one. After the loop ends, it closes the file by invoking the `Close` method on the `StreamReader` object.

This code **does not** manage the file resource correctly.

If an exception is thrown in the `ProcessLine` method, the “fileReader.Close()” statement will not execute and the file will remain open. If we then try to write to the same file, it will fail with an IOException saying that the file cannot be accessed because it is currently being used.

In .NET, we can easily fix this problem by using the file inside a `try` block and then closing it in a corresponding `finally` block. This pattern is so common that in C# we have a special statement, the `using` statement, that we can use to create the effect of closing (or Disposing in more formal terminology), the resource in a `finally` block.

Here is an example:

```
using (var fileReader = new StreamReader(“c:\\file.txt”))
{
  while (true)
  {
    var line = fileReader.ReadLine();

    if (line == null)
      break;

    ProcessLine(line);
  }
}
```

This works because the `StreamReader` class implements the `IDisposable` interface. Before execution leaves the using block, it is guaranteed that the `Dispose` method of the `StreamReader` object will be called and the file will be closed even if an exception is thrown. For more information about the `IDisposable` interface and the using statement, see this MSDN reference (bit.ly/dnc-dispose).

Another issue related to resource management is the **efficient use of resources**.

One way we can use a resource, say a HTTP connection (the underlying TCP connection), is to open the connection, invoke some HTTP request, receive the response, and then close the connection. If we need to send 10 HTTP requests, we would open a new connection ten times and close it an equal number of times.

It makes more sense however to open the connection, use it to send multiple HTTP requests for some period of time and then close it.

This is true because opening a connection is expensive!!

Consider HTTPS (HTTP with SSL/TLS) connections for example. Every time we open a new HTTPS connection, the client needs to authenticate the server and sometimes even the server requires client authentication. Also, the two parties need to agree on a symmetric key to use to protect the session.

When working with resources in the scope of a single method, handling the two mentioned issues, is easy. For the first issue, we can use the **try...finally** syntax (or the **using** statement) to make sure we always close resources. And for the second issue, we can open the resource, use it many times, and then close it.

However, when a resource needs to be used from multiple objects in a complex application, these issues become harder to handle.

In order to efficiently use the resource, the application objects need to share the same resource instance. Also, some resources need to be shared by multiple objects because they represent something unique, e.g. a particular file on the file system.

If resources are shared between multiple objects, how can we make sure that they are closed when they are no longer needed? Who is responsible for closing the resource and when? Who is responsible for opening the resource in the first place?

Before going forward, let's first describe a resource in a somewhat formal way. A resource (in this article) refers to something that:

1. Needs to be opened before used
2. Opening it is expensive
3. Needs to be closed after we are done using it
4. Might represent something unique that needs to be shared

Resources versus Single-method operations

Before going into how to deal with resources in the scope of large applications, let's first consider the difference between how we deal with resources and how we deal with single-method operations.

In our applications, we would like to have operations that are represented by single methods. For instance, we would like to have a method called `CalculateDiscountForCart` that takes a cart data object and calculates how much discount we should apply. This method by itself represents a useful operation that we can execute.

When dealing with resources on the other hand, we need to invoke multiple methods.

For example, we need to invoke a method to open the resource, another one to use it, and another one to close it.

Consider the following example:

```
public interface ICartDiscountCalculator
{
  decimal CalculateDiscountForCart(Cart cart);
}
```

This interface has a single method, `CalculateDiscountForCart`. Consumers need only to worry about this single method.

Now consider the following example of some resource:

```
public interface IResource
{
  void OpenConnection();
  string QuerySomeData();
  void CloseConnection();
}
```

Now the consumer of this interface needs to worry about three methods. It needs to invoke them in a particular order, and it also needs to make sure that the `CloseConnection` method is called if the `OpenConnection` method was called.

It is a lot easier to deal with contracts that have a single method (or contracts that have multiple methods but that are completely independent). Therefore, in the rest of the article, I will discuss solutions to resource management problems that will allow the direct consumers of the resource to only have to deal with a single method, e.g. `QuerySomeData`.

Now let's consider a simple approach to managing resources in a large application.

Managing Resources - A simple approach

Let's first consider an approach to resource management that handles resources correctly, provides a simple interface for consumers, but might be inefficient.

A simple way with which we can handle a resource is open and close it every time we need to use it.

As discussed before, in a complex application, we should hide this behind a simple interface so that other code blocks can use it without the need to care about any resource management.

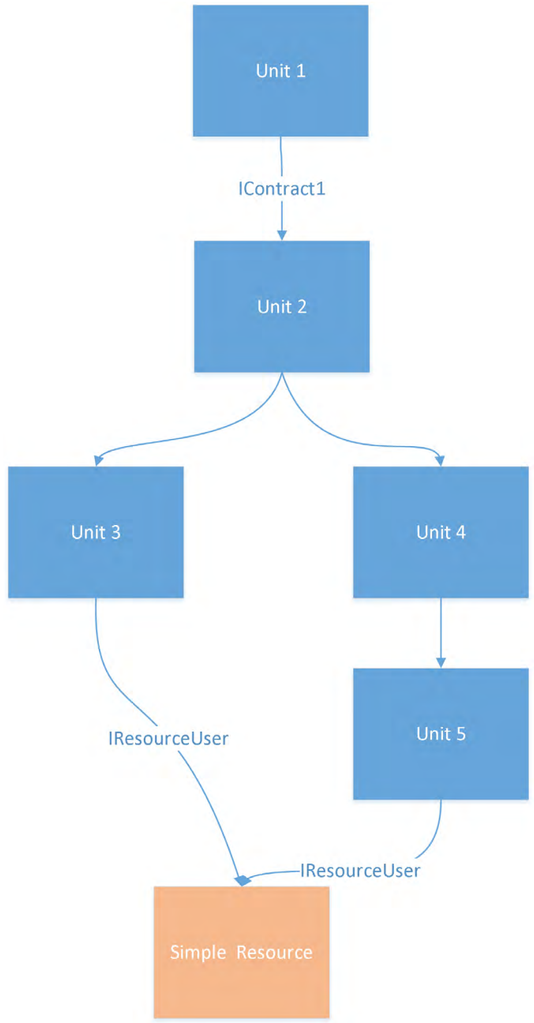
Consider this example:

```
public interface IResourceUser
{
  string QuerySomeData();
}

public class SimpleResource : IResourceUser
{
  public string QuerySomeData()
  {
    //Open some resource
```

```
    try
    {
      //Query the resource
    }
    finally
    {
     //Close the resource
    }
  }
}
```

Now, the consuming code only needs to know about the `QuerySomeData` method.

Consider the following object graph:



In this figure, Unit 3 and Unit 5 are the direct consumers of a `SimpleResource` object. They are connected to this object via the `IResourceUser` interface. They use it to call the `QuerySomeData` method.

Unit 1 (which in this figure is the starting point, e.g. a controller in a Web API application) has a dependency on IContract1. In this figure, it is connected to an instance of the Unit2 class. Unit 1 invokes Unit 2, which in turn invokes Unit 3, Unit 4 and eventually Unit 5.

Here is how we can compose such a graph in the Composition Root:

```
var resource = new
SimpleResource(resourceAddress);

var instance =
  new Unit1(
    new Unit2(
      new Unit3(resource),
        new Unit4(
          new Unit5(resource))));
```

Each time Unit 3 or Unit 5 uses the resource, the resource is opened, used, and then closed.

If Units 3 and 5 call `QuerySomeData` 10 times, this means that we will open and close the resource 10 times.

For some scenarios, this is perfectly acceptable.

But for others, this solution is inefficient.

How can we have the consumers know only about `QuerySomeData`, but still be able to use the resource multiple times before we close it?

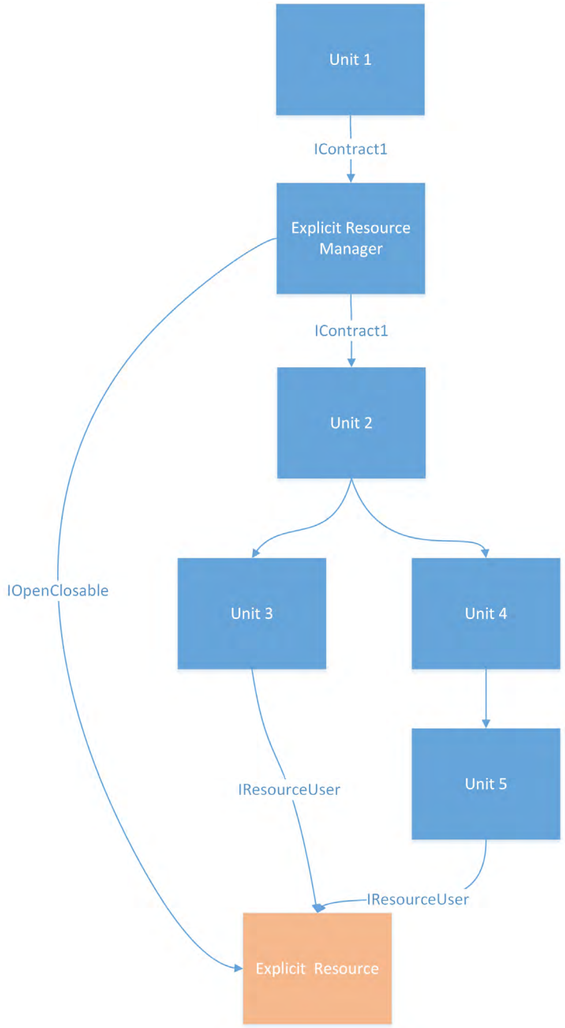
Explicit management

To solve the issue just mentioned, we can change the resource class to have explicit `Open` and `Close` methods like this:

```
public interface IResourceUser
{
  string QuerySomeData();
}

public interface IOpenClosable
{
  void Open();

  void Close();
}

public interface IResource : IResourceUser, IOpenClosable
{
}

public class ExplicitResource : IResource
{
  //Handle to resource is stored in a private field

  public string QuerySomeData()
  {
    //Query the resource
  }

  public void Open()
  {
     //Open the resource
  }

  public void Close()
  {
    //Close the resource
  }
}
```

The direct consumer of the resource needs to have a dependency only on `IResourceUser` because it only needs to invoke `QuerySomeData`. However, someone else needs to call `Open` and `Close`. That someone needs to call `Open` before the direct consumers of the resource have the chance to invoke `QuerySomeData`, and it needs to call `Close` after the direct consumers are done using the resource.

Consider the following updated object graph:



Unit 1 is now connected to an instance of an `ExplicitResourceManager` class. This class is simply a decorator for IContract1. It intercepts the call between Unit 1 and Unit 2 and opens the resource by invoking the `Open` method on a dependency of type `IOpenClosable` (which is connected to the same resource instance as Unit 3 and 5).

When execution eventually goes back to the ExplicitResourceManager unit, it closes the resource via `IOpenClosable.Close.`

Here is how we compose this graph in the Composition Root:

```
var resource = new
ExplicitResource(resourceAddress);

var instance =
  new Unit1(
    new ExplicitResourceManager(
      new Unit2(
        new Unit3(resource),
          new Unit4(
            new Unit5(resource))),
            openClosable: resource));
```

So far, we have achieved two goals:

1. Direct consumers of the resource need only to deal with a single method.

2. The resource can be used multiple times before it is closed.

We can still modify our approach to solve the following issues:

1. The resource might be opened before it is really needed. For example, in the previous graph, the resource is opened immediately before Unit 2 is invoked. What if there are some 10 seconds before Unit 2 being invoked and Unit 3 and 5 deciding to use the resource?

2. What if Unit1 is invoked 100 times per second? This means that we open the resource and close it 100 times per second (although each time we might use it a few times). Wouldn't it be better if we open the resource once, use it for 100 invocations of Unit 1, and then close it?

To solve the first issue, we can make the `ExplicitResourceManager` unit responsible only for closing the resource and then make the `ExplicitResource` object auto-open when `QuerySomeData` is invoked. To do so, the state inside the `ExplicitResource` object needs to keep track of whether the resource is currently opened.

In the next section, we will build on this to solve the second issue.

Automatic resource management

Another approach to manage resources is to automatically open and close them.
We can make the resource open itself before it is used (like we did in the simple approach), and then have it close automatically after some predefined period of idle time.

Take a look at the AutomaticManager class here. This class implements the IResourceUser interface and therefore its consumers need to care only about a single method, the `QuerySomeData` method.
This class has a dependency on IResource. It also has a configuration parameter, `maximumIdleTimeBeforeClosing` of type `TimeSpan`. This parameter will allow us to configure how much time the resource is allowed to be idle, before it is automatically closed.

In the `QuerySomeData` method, we first check to see if the resource is already opened and open it if needed. We then invoke the `QuerySomeData` method on the resource. Then, in the finally block, we want to create a timer that would close the resource after some idle time.

To do so, we do the following:

1. We keep track of the last time a request (a call to `QuerySomeData`) is made by maintaining a stopwatch, `lastRequestStopwatch`, and restarting it every time the resource is used.

2. We keep track of the number of requests made since the request was opened, via the `requestNumber` field.

3. If `requestNumber` is 1, i.e., if this is the first request since the resource was opened, then we schedule the closing of the resource in the future. We do this inside the `ScheduleResourceClosing` method. More specifically, in the following line:

```
await Task.Delay(delay).ConfigureAwait(false);
```

..we ask the system to asynchronously wait for some time (specified by the `maximumIdleTimeBeforeClosing` parameter). The calling thread will return immediately when "await" is executed. When the timer elapses (the task returned by `Task.Delay` is completed), the `ScheduleResourceClosing` method will continue execution on a thread pool thread because of the `ConfigureAwait(false)` call.

If `requestNumber` is still 1, then this means that no more requests were sent while we were waiting, and therefore, we simply close the resource, reset the stopwatch, and the `requestNumber` variable.

If `requestNumber` is not 1, then this means that while we were waiting, other requests were made. We need to wait more so that we end up waiting `maximumIdleTimeBeforeClosing` from the time of the last request.

This is the reason of the loop inside the `ScheduleResourceClosing` method. We calculate a new value of *delay* that takes into account the time that has passed since the last request. This is why we keep track of the time of the last request via the `lastRequestStopwatch` stopwatch.

In Program.cs, in the `TestAutomaticManager` method, I have written some code to test this:

```
IResourceUser resourceUser =
  new AutomaticManager(
    new FakeResource(),
      TimeSpan.FromMilliseconds(500));

resourceUser.QuerySomeData();
Thread.Sleep(400);
resourceUser.QuerySomeData();
Thread.Sleep(600);
resourceUser.QuerySomeData();
Console.WriteLine("Done");
Console.ReadLine();
```

This code first creates a FakeResource object. This object simply prints to the console when the Open, Close, or QuerySomeData methods are invoked.

The code then wraps this fake resource object with an instance of the `AutomaticManager` class specifying a value of 500 milliseconds for `maximumIdleTimeBeforeClosing`.

We then invoke `QuerySomeData` three times. Intentionally, we wait different periods of time between every two invocations of `QuerySomeData` to test the behavior of `AutomaticManager`.

This is what is printed to the console when the application executes:

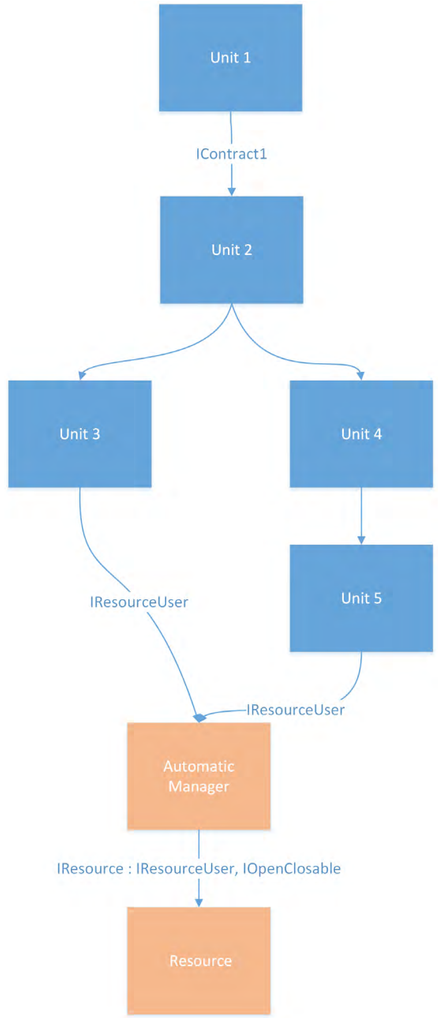
Opening the resource
Querying the resource
Querying the resource
Closing the resource
Opening the resource
Querying the resource
Done
Closing the resource

When we call `QuerySomeData` for the first time, the resource is opened. We see two messages in the console indicating that the resource was opened and then queried.

Then after we wait 400 milliseconds and invoke the method again, the resource is queried and a message in the console is printed to indicate this. No messages to indicate that the resource is opened or closed are printed, as expected.

Then we wait 600 milliseconds. Before waiting is complete, a message indicating that the resource was closed is printed. This is because the resource auto-closes after 500 milliseconds.
In order to service the third request, the resource is opened again. We see two messages indicating the opening and the querying of the resource.

We then see the "Done" message printed to the console.
After 500 milliseconds, the resource auto-closes again and a closing message is printed to the console.
Here is how the graph for the imaginary example used earlier would look like after we use this approach:



This graph is very similar to the simple approach graph. We don't need any objects up in the graph to manage the resource explicitly.

The automatic management aspect

In a real application, you might have many resources, each with different interfaces, and many with multiple usage/querying methods.

The `AutomaticManager` class works only with the example `IResource` and `IResourceUser` interfaces. We can do the same thing for other resource interfaces by creating an aspect.

For more information about aspects, see the Aspect Oriented Programming (AOP) in C# with SOLID article (bit.ly/dnc-aop-solid).

Take a look at the AutomaticManagerAspect class which is very similar to the `AutomaticManager` class. This aspect is built using the DynamicProxy library.

Also take a look at the `TestAutomaticManagerAspect` method in the Program.cs file to see how to use it. The aspect expects a dependency of type IOpenClosable in the constructor. You can use such an interface in your resource classes for the `Open` and `Close` methods. If you can't, then you can give the aspect an adapter object that adapts your resource object to IOpenClosable.

Customizing the automatic manager

The approach described above can be modified and customized in many ways.

For example, currently the automatic management classes assume that the resource can only be used by a single thread. You might encounter a scenario where you want the same resource instance to be used by multiple threads.

Or you can have a *pool* of resources managed automatically by the automatic manager.

For example, you can configure the manager to have a pool of at most three instances of the resource.
If three concurrent threads want to use the resource, each of them will get a different instance. A fourth thread that wants to use the resource would have to wait for one of the other three threads to finish using the resource.

Another customization is to enable the resource to auto-renew after some time.

For example, you might want a connection to be closed and opened again every ten minutes. Or you might customize it to make it renew at 12:00 AM every day.

Or you might want the resource to auto-renew every twenty invocations, e.g. after `QuerySomeData` is called twenty times.

Existing automatic resource management in the .NET framework

For performance reasons, the .NET framework has some automatic management features built into it.
For example, if you use the `HttpClient` class to send multiple requests to some HTTP server from a single thread, a single connection will be used. The connection will only be closed after some time elapses without any requests being sent. The mechanics of this feature are different from those of the approach I described above, but the general idea is the same.

The consumer of the `HttpClient` class does not need to worry about opening and closing the connection. All the consumer needs to do is invoke methods that use the client like HttpClient.GetAsync, or HttpClient.PostAsync, etc.

This is very similar to the idea I discussed in the article where the consumer needs to only worry about using the resource, not opening and closing it. Although the `HttpClient` class implements the IDisposable interface, it is a recommended practice that developers create a single instance of it and use it for the whole duration of the application, i.e., they don't need to dispose it. See this blog post (bit.ly/dnc-http-best) for more details.

Another example of automatic resource management in the .NET framework is related to ADO.NET and SQL Server.

When using ADO.NET to connect to SQL Server, ADO.NET manages a pool of connections to the SQL server. This means that when you open and close a `SqlConnection` object, the real connection remains open for some time. If you later (within some frame of time) open another `SqlConnection` object with the same connection string, you would use the same real connection to the SQL server.

Again, the specifics of this feature are different from the approach I described in the article, but the general idea is the same. For more information about SQL Server Connection Pooling see this article here.

One particular thing to note about `SqlConnection` object is that you still need to "Close" the connection for it to be marked as not used. You also need to explicitly "Open" the connection to retrieve an existing connection from the pool or to actually create a new one.

This is different from the `HttpClient` scenario and the automatic management example I described in the article. In both of these cases, the consumer doesn't need to invoke "Open" or "Close".

As you can see, the .NET framework already does a lot of work to make sure many kinds of resources are managed in a somewhat automatic way so that developers don't need to worry about doing such management themselves.
Then when should a developer use the approaches described in this article?

Resource management Examples

In general, you can use the approaches described in this article whenever you work with resources that meet the definition I provided in the beginning of the article. These resources do not have any built-in automatic management features, and you need to use them from multiple objects in the application.

Let's consider some examples.

The SmtpClient class

The SmtpClient class in the .NET framework allows us to send email messages. We can use the `Send` method to send messages.

Internally, the `SmtpClient` class implementation pools SMTP connections. This means that when we invoke `Send` to send a message, an existing connection might be used.

This is great.

However, the `SmtpClient` class does not send the QUIT command to end existing SMTP sessions (to free server resources) until we explicitly call the `Dispose` method. This `Dispose` method will not be called automatically after some time of inactivity (or after some other condition as described in the Customizing the automatic manager section).

We can easily encapsulate the `SmtpClient` class inside some `SmtpResource` object. The `Open` method would create an instance of the `SmtpClient` class, and the `Close` method would call `Dispose`.

We should also provide some methods to send the email messages.

We can then use the `AutomaticManagerAspect` to auto-manage this resource allowing multiple objects to use the SMTP client resource without worrying about disposing/closing anything. We can make the resource close after some period of time of inactivity, or maybe after twenty messages have been sent.

A locked file

Another example could be when we might want to log some sensitive information to some file in some folder.

Each day we create a new file in this folder and open it for writing, and for security reasons we specify that we don't want anyone else to be able to access the file. To do so, we specify the `FileShare.None` option when we create and open the file, for example via the File.Open method.

The file also needs to be signed before it is closed. This means that special code needs to run before we close the file.
Multiple objects in the application graph need to access the resource object that encapsulates the `FileStream` returned by the `File.Open` method.

The Open method of the resource object would first determine the new filename to use (e.g. generate a random file name or generate a name using the current time), and then call `File.Open` and store the `FileStream` in some private field.

The resource would declare some `Log` method that will be used by the direct consumers.

The `Close` method would sign the content of the file and then close it.

In this example, the resource is the unique file that needs to be shared by multiple objects in the graph.

Third part libraries

Many third-party libraries allow you to access resources, e.g. an SMTP server. They provide some Open and Close methods but they don't have any built in automatic management features (e.g. pooling). In this case, you can use the approaches described in this article to encapsulate the resources provided by these libraries to automate the process of managing them.

Conclusion:

In this article I discussed the issues we usually face when we deal with resources.

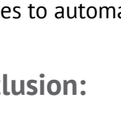
Resources are things that we cannot simply invoke like ordinary single-method operations. A resource needs to be opened, used, and eventually closed. We need to make sure it is always closed even in case of error.

Also, for performance reasons, we might want to open the resource, use it multiple times, and then close it.

When using a resource from multiple objects in an application, dealing with resources becomes a bit harder.

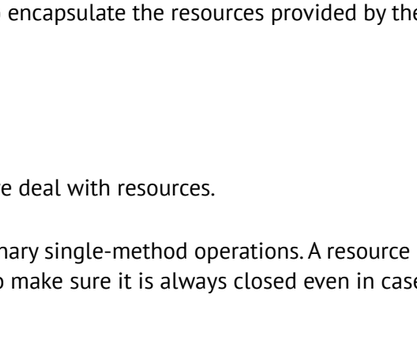
In this article I discussed some approaches on how to deal with them in this scope. These approaches allow the consumers of a resource to use it without needing to worry about opening and closing it.

I also gave examples of automatic resource management that exists in the .NET framework and described some examples of when someone might decide to implement an automatic management solution



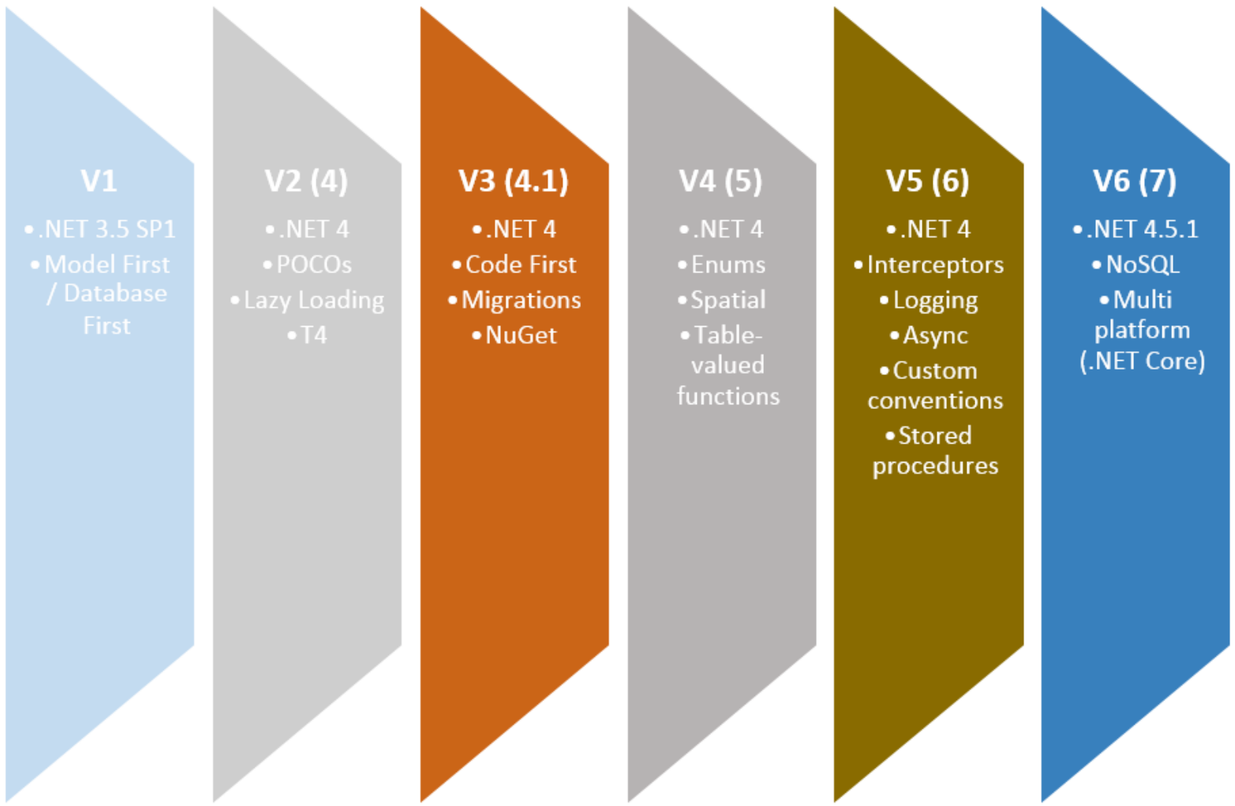
Yacoub Massad
Author

Yacoub Massad is a software developer who works mainly with Microsoft technologies. Currently, he works at Zeva International where he uses C#, .NET, and other technologies to create eDiscovery solutions. He is interested in learning and writing about software design principles that aim at creating maintainable software. You can view his blog posts at criticalsoftwareblog.com



Thanks to Damir Arh for reviewing this article.

Ricardo Peres



The State of Entity Framework Core

Entity Framework 1 was introduced back in 2008 with .NET 3.5 Service Pack 1. During the same time, LINQ to SQL also came out, but Microsoft didn't consider it to be an Object-Relational Mapper (O/RM). Unlike Entity Framework, LINQ to SQL was only meant to support SQL Server.

It featured both LINQ, a brand new toy back then – and something called Entity-SQL, an object-oriented SQL-alike dialect that never really caught up. It had features like multiple inheritance strategies, compiled queries, explicit loading, change tracking and other features that are commonly expected from an ORM. It was based on a provider model, so in theory it was able to support any relational database for which there was an ADO.NET Provider. Visual Studio could reverse-engineer a database and produce a model for it.

However it was received with mixed feelings. On one end, it was good to see Microsoft having a go at ORM, but there were already other products that could do the same and more. NHibernate had been around for some time and was quite popular back then.

The problems with LINQ to SQL were:

- Although it was touted that it supported other databases, it really only worked well with SQL Server
- It was complex to configure in Visual Studio – let alone XML configuration
- Entities had to inherit from a base class
- The generated SQL was far from optimal

Some developers even issued a vote of no confidence (http://efvote.wufoo.com/forms/ado-net-entity-framework-vote-of-no-confidence) to show how bad they felt about it.

The second version, now bumped to v4 to align with the release of .NET 4 in 2010 improved it a bit – with self-tracking entities with an eye for WCF, lazy loading, initial support for Plain-Old CLR Objects (POCOs), but people were still unsatisfied, even though it was then Microsoft's recommended database access technology.

It was only with version 4.1, popularly known as Code First, which happened in 2011, that things started to look interesting.

Entity Framework went through a big refactor, it seemed much more polished and clean, fully embracing POCOs and was code-centric. It was now much easier to start working with. Almost no mapping and configuration was required as it was based on conventions – things just worked out of the box. The version was somewhat misleading, as it seemed to be a whole new version, but underneath, the "old" Entity Framework was still there.

Another version 4.3, came out shortly which featured migrations, a welcome addition, but not one without its problems.

Then came version 5, bringing support for enumerated and spatial types and table-valued functions, and was released in the same year, 2012.

Finally, version 6 brought most of the stuff that mature ORMs support, such as interceptors, logging capabilities, custom conventions, support for stored procedures and asynchronous methods.
Life was good in 2013!

But then Microsoft started working on something new, and this had a profound impact on Entity Framework. Enter .NET Core!

ENTITY FRAMEWORK CORE

.NET Core was in the forge for several years, and it's not hard to understand why.

Put it simply, Microsoft wanted to rewrite its framework from scratch, make it open-source and support operating systems other than Windows – not something to be taken lightly!

Of course, Entity Framework had to be re-implemented as well, and Microsoft took the opportunity to do things right this time. EF Core, as it is now called, was totally rewritten, but tried to stay close to Code First.

Because it was a massive task, a roadmap was defined for its features, but from the start, a couple of them were received with lot of interest:

- It would work on all operating systems (Linux, Mac) and platforms (Windows Phone, Windows Store) supported by .NET Core

- Non-relational databases would also be supported, making it not just an ORM: Azure Table Storage and Redis were the first to be announced

- Fully extensible and provider-based

The Entity Framework Core team reunites regularly and makes available a summary of their decisions. Interested folks can submit issues and participate in the discussions, but of course, the last word belongs to Microsoft.

EF Core 1 was eventually released in 2016, followed by version 1.1 the same year, but it was far from what was expected.

EF CORE - THE GOOD BITS

The good about it was:

- Brand new code base, with most types public and following best practices, all available in GitHub

- It did work on operating systems other than Windows and on platforms other than PCs

- Almost infinitely extensible

- Support for providers other than SQL Server (SQL Server Compact Edition, SQLite, InMemory, PostgreSQL, MySQL, DB2, MyCat, Oracle)

- A simplified API that is close to the previous one (Code First)

- New features: shadow properties, ability to mix SQL with LINQ, transparent client-side function evaluation, high-low identifier generation, batching of queries, improved SQL generation

EF Core v2.0 has just been released and it brought along other interesting features, chief among them are global queries (very handy for multi-tenant and soft-deletes), owned types (complex values) and a limited form of support for database functions.

To summarize, EF Core 2 is pleasant to use. It's easy to get started with.

The shortcoming needs some explaining.

EF CORE 2.0 - SHORTCOMINGS

1. One of the more appealing new features was the ability to target non-relational databases.
It seems like a great idea – just use the same APIs and LINQ to work with potentially any kind of data source imaginable.

The problem is, this is yet to be seen, as no such provider was actually released – if you don't consider InMemory of course! The code is nowhere to be found on GitHub and we have no idea of what it will be like, when (if?) it is actually released.

2. Microsoft boasts about improvements in the SQL generation in EF Core, but a crucial feature such as support for grouping (LINQ's GroupBy operator) is not supported yet – or worse, it does not fail, yet silently does it all in memory, with possibly a tremendous impact on performance.

Unaware developers may have a bad time with this, and there is no easy solution to it, essentially because it is not possible to project to non-entity classes yet.

3. Worse, it is not possible to perform date and time operations with LINQ, and lots of SQL standard mathematical operators are also missing. Version 2 brought support for LIKE, but that's it.

4. Lazy loading is a feature that people are expected to use, but alas, it is not here. Version 1.1 brought explicit loading, and eager loading was here since version 1, but it's still surprisingly absent.

5. No many-to-many collections and no inheritance strategies, only single table inheritance.

6. Migrations are here already, but a missing feature is database initializers or the capability to seed the database. We have to rely on custom SQL for that, which is at least cumbersome, when we're talking about an ORM.

7. No interception capabilities exist yet, meaning, it's very difficult to modify a query before it is actually executed. No out of the box way to delegate entity creation and initialization.

8. The ability to use stored procedures for Create-Update-Delete (CUD) operations is also not yet implemented. It is possible to use stored procedures for querying, though.

9. It is not possible yet to plug our own custom conventions.

10. Spatial types are absent too, but enumerations and complex types are now supported.

11. Also, no support for System.Transactions (aka, TransactionScope) and distributed transactions. I guess we can understand that since the first steps were to get this running on Linux and Mac, for which there is no Microsoft Distributed Transaction Coordinator (MS DTC), this is a feature that people are used to.

12. Finally, still no Visual Studio (VS) integration. It is not possible to reverse-engineer a database from inside VS or to apply database migrations, except of course, using Package Manager.

From what I mentioned, it should be clear that EF Core is not ready for enterprise-level usage, in fact, Microsoft even says that on the Roadmap page.

It is obvious that EF Core is new and Microsoft hasn't had the time to implement everything, but some of the absences – grouping, date/time and mathematical operations, lazy loading – are hard to understand.

Also, developers had shown a deep interest for non-relational providers and these are conspicuously absent.

ALTERNATIVES TO EF CORE

There are few full-featured ORMs that can work with .NET Core, one of which is LLBLGen Pro, but it is a commercial product.

NHibernate still does not target .NET Core or .NET Standard, and by the looks of things, it will take a long time until it does. Dapper, the micro-ORM, is already available for .NET Core, but it is not a full-featured product. Clearly, there is no free cross-platform alternative to EF Core yet, so with all its shortcomings, it's probably your best option for now.

If we are to compare Entity Framework Core vCurrent with another well-established O/RM, NHibernate, we can see that EF is still behind in a number of features including:

- Lots of identifier generation strategies
- Different collection types, including collections of primitive types
- Database-independent dialect for querying and doing batch updates (HQL)
- Support for non-trivial mapping scenarios
- Lazy properties
- Type conversions

The problem with NHibernate currently seems to be the lack of interest from the community, even if some of its features are still ahead of its competitors.

WHAT'S NEXT

The Entity Framework Core Roadmap page (https://github.com/aspnet/EntityFrameworkCore/wiki/Roadmap) details what are the features that will be implemented in the next version, 2.1.

We can see that some of the missing features I highlighted – grouping, lazy loading (probably), System. Transactions, lifecycle events – are already there/planned, plus a few others like support for Oracle and Cosmos DB.

It is possible that things are going in the right direction. There are however some features, like mapping strategies, concrete table inheritance and class table inheritance, who don't seem to be attracting a lot of interest from the community. So probably these will not be around for quite some time.

We, as developers, have a saying in this, however Microsoft this time seems to be listening to the community, so it's our responsibility to bring our concerns to them, and contribute to their resolution.

Conclusion

Entity Framework Core is cool because it can run on any of the .NET Core supported platforms and it offers some interesting features over the previous, pre-Core, versions.

It is quite new, so it is still lacking lots of stuff, but since the community is helping, these features may actually be implemented sooner than one would think.

I think EF is getting interest from developers, although people were somewhat unhappy with the absence of some functionality that are deemed essential by some. For now, as there is no clear alternative for multi-platform work, you may want to give it a try

Microsoft MVP Most Valuable Professional

Ricardo Peres
Author

Ricardo Peres is a Portuguese developer, blogger, technical reviewer and an occasional book author. He has more than 17 years of experience in software development, using technologies such as C/C++, Java, JavaScript, and .NET. His interests include distributed systems, architectures, design patterns, and general .NET development.

He currently works for London-based Simplifydigital as a Technical Evangelist, and is a Microsoft MVP in Visual Studio and Development Technologies. Ricardo maintains the blog Development With A Dot, where he regularly writes about technical issues. You can read it at http://weblogs.asp.net/ricardoperes. You can catch up with him on Twitter as @rjperes75.

Thanks to Suprotim Agarwal for reviewing this article.

David Pine

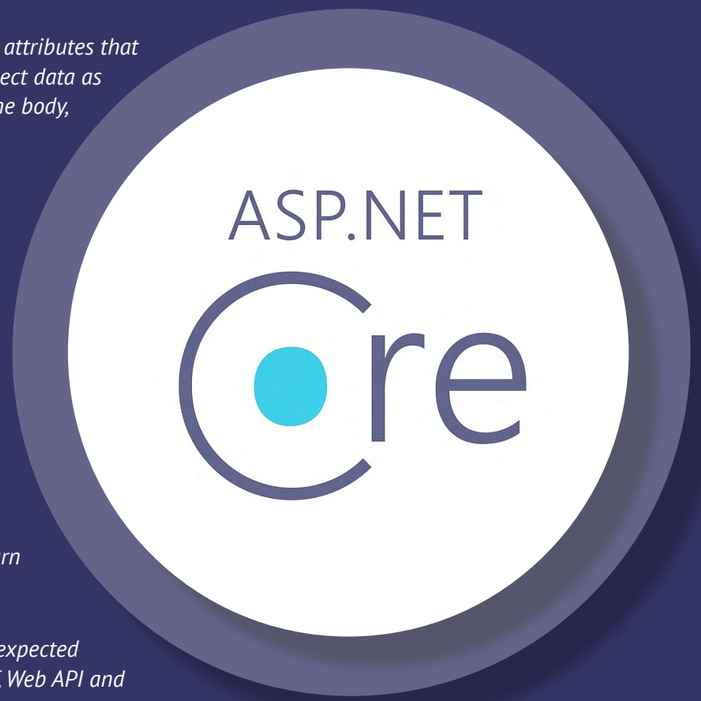
ASP.NET Core
Web API Attributes

With ASP.NET Core there are various attributes that instruct the framework where to expect data as part of an HTTP request - whether the body, header, query-string, etc.

With C#, attributes make decorating API endpoints expressive, readable, declarative and simple. These attributes are very powerful! They allow aliasing, route-templating and strong-typing through data binding; however, knowing which are best suited for each HTTP verb, is vital.

In this article, we'll explore how to correctly define APIs and the routes they signify. Furthermore, we will learn about these framework attributes.

As a precursor to this article, one is expected to be familiar with modern C#, REST, Web API and HTTP.



ASP.NET Core has HTTP attributes for seven of the eight HTTP verbs listed in the Internet Engineering Task Force (IETF) RFC-7231 Document. The HTTP TRACE verb is the only exclusion in the framework. Below lists the HTTP verb attributes that are provided:

- HttpGetAttribute
- HttpPostAttribute
- HttpPutAttribute
- HttpDeleteAttribute
- HttpHeadAttribute
- HttpPatchAttribute
- HttpOptionsAttribute

Likewise, the framework also provides a `RouteAttribute`. This will be detailed shortly. In addition to the HTTP verb attributes, we will discuss the *action* signature level attributes. Where the HTTP verb attributes are applied to the action, these attributes are used within the parameter list. The list of attributes we will discuss are listed below:

- FromServicesAttribute
- FromRouteAttribute
- FromQueryAttribute
- FromBodyAttribute
- FromFormAttribute

Ordering System

Imagine if you will, that we are building out an ordering system. We have an order model that represents an order. We need to create a *RESTful* Web API that allows consumers to create, read, update and delete orders – this is commonly referred to as CRUD.

Operation	Verb	API Endpoint
CREATE	HTTP POST	api/orders
READ	HTTP GET	api/orders/{id}
UPDATE	HTTP PUT	api/orders/{id}
DELETE	HTTP DELETE	api/orders/{id}

Route Attribute

ASP.NET Core provides a powerful `Route` attribute. This can be used to define a top-level route at the controller class – doing so leaves a common route that actions can expand upon. For example consider the following:

```
[Route("api/[Controller]")]
public class OrdersController : Controller
{
  [HttpGet("{id}")]
  public Task<Order> Get([FromRoute] int id)
  => _orderService.GetOrderAsync(id);
```

The `Route` attribute is given a template of `"api/[Controller]"`. The `"[Controller]"` is a special naming convention that acts as a placeholder for the controller in context, i.e.; `"Orders"`. Focusing our attention on the `HttpGet` we can see that we are providing a template argument of `"{id}"`. This will make

the HTTP Get route resemble “api/orders/1” – where the id is a variable.

HTTP GET Request

Let us consider an HTTP GET request. In our collection of orders, each order has a unique identifier. We can walk up to the collection and ask for it by “id”. Typical with *RESTful* best practices, this can be retrieved via its route, for example “api/orders/1”. The action that handles this request could be written as such:

```
[
  HttpGet(“api/orders/{id}”) // api/orders/7

]
public Task<Order> Get(
  [FromRoute] int id,
  [FromServices] IOrderService orderService)
  => orderService.GetOrderAsync(id);
```

Note how easy it was to author an endpoint, we simply decorate the controller’s action with an HttpGet attribute.

This attribute will instruct the ASP.NET Core framework to treat this action as a handler of the HTTP GET verb and handle the routing. We supply an endpoint template as an argument to the attribute. The template serves as the route the framework will use to match on for incoming requests. Within this template, the {id} value corresponds to the portion of the route that is the “id” parameter.

This is a Task<Order> returning method, implying that the body of the method will represent an asynchronous operation that eventually yields an Order object once awaited. The method has two arguments, both of which leverage attributes.

First the FromRoute attribute tells the framework to look in the route (URL) for an “id” value and provide that as the id argument. Then the FromServices attribute – this resolves our IOrderService implementation. This attribute asks our dependency injection container for the corresponding implementation of the IOrderService. The implementation is provided as the orderService argument.

We then expressively define our intended method body as the order services’ GetOrderAsync function and pass to it the corresponding identifier.

We could have just as easily authored this to utilize the FromQuery attribute instead. This would then instruct the framework to anticipate a query-string with a name of “identifier” and corresponding integer value. The value is then passed into the action as the id parameters argument. Everything else is the same.

However, the most common approach is the aforementioned FromRoute usage – where the identifier is part of the URI.

```
[
  HttpGet(“api/orders”) // api/orders?identifier=7
]
public Task<Order> Get(
  [FromQuery(Name = “identifier”)] int id,
  [FromServices] IOrderService orderService)
  => orderService.GetOrderAsync(id);
```

Notice how easy it is to alias the parameter?

We simply assign the Name property equal to the string “identifier” of the FromQuery attribute. This instructs the framework to look for a name that matches that in the query-string. If we were to omit this argument, then the name is assumed to be the name used as the actions parameter, “id”. In other words, if we have a URL as “api/orders?id=17” the framework will not assign our “id” variable the number 17 as it is explicitly looking for a query-string with a name “identifier”.

HTTP POST Request

Continuing with our ordering system, we will need to expose some functionality for consumers of our API to create orders.

Enter the HTTP POST request.

The syntax for writing this is seemingly identical to the aforementioned HTTP GET endpoints we just worked on. But rather than returning a resource, we will utilize an IActionResult. This interface has a large set of subclasses within the framework that are accessible via the Controller class. Since we inherit from Controller, we can leverage some of the conveniences exposed such as the StatusCode method.

With an HTTP GET, the request is for a resource; whereas an HTTP POST is a request to create a resource and the corresponding response is the status result of the POST request.

```
[
  HttpPost(“api/orders”)
]
public async Task<IActionResult> Post([FromBody] Order order)
  => (await _orderService.CreateOrderAsync(order))
  ? (IActionResult)Created($”api/orders/{order.Id}”, order) // HTTP 201
  : StatusCode(500); // HTTP 500
```

We use the HttpPost attribute, providing the template argument.

This time we do not need an “{id}” in our template as we are being given the entire order object via the body of the HTTP POST request. Additionally, we will need to use the async keyword to enable the use of the await keyword within the method body.

We have a Task<IActionResult> that represents our asynchronous operation. The order parameter is decorated with the [FromBody] attribute. This attribute instructs the framework to pick the order out from the body of the HTTP POST request, deserialize it into our strongly-typed C# Order class object and provide it as the argument to this action.

The method body is an expression. Instead of asking for our order service to be provided via the FromServices attribute like we have demonstrated in our HTTP GET actions, we have a class-scope instance we can use. It is typically favorable to use constructor injection and assign a class-scope instance variable to avoid redundancies.

We delegate the create operation to the order services’ invocation of CreateOrderAsync, giving it the order. The service returns a bool indicating success or failure. If the call is successful, we’ll return an HTTP status code of 201, Created. If the call fails, we will return an HTTP status code of 500, Internal Server Error.

Instead of using the FromBody one could just as easily use the FromForm attribute to decorate our order parameter. This would treat the HTTP POST request differently in that our order argument no longer comes from the body, but everything else would stay the same. The other attributes are not really applicable with

an HTTP POST and you should avoid trying to use them.

```
[
  HttpPost(“api/orders”)
]
public async Task<IActionResult> Post([FromForm] Order order)
  => (await _orderService.CreateOrderAsync(order))
    ? Ok() // HTTP 200
    : StatusCode(500);
```

Although this bends from HTTP conformity, it’s not uncommon to see APIs that return an HTTP status code *200, Ok* on success. I do not condone it.

By convention if a new resource is created, in this case an order, you should return a 201. If the server is unable to create the resource immediately, you could return a *202, accepted*. The base controller class exposes the `Ok()`, `Created()` and `Accepted()` methods as a convenience to the developer.

HTTP PUT Request

Now that we’re able to create and read orders, we will need to be able to update them. The HTTP PUT verb is intended to be idempotent. This means that if an HTTP PUT request occurs, any subsequent HTTP PUT request with the same payload would result in the same response. In other words, multiple identical HTTP PUT requests are harmless and the resource is only impacted on the first request.

The HTTP PUT verb is very similar to the HTTP POST verb in that the ASP.NET Core attributes that pair together, are the same. Again, we will either leverage the `FromBody` or `FromForm` attributes. Consider the following:

```
[
  HttpPut(“api/orders/{id}”)
]
public async Task<IActionResult> Put([FromRoute] int id, [FromBody] Order order)
  => (await _orderService.UpdateOrderAsync(id, order))
    ? Ok()
    : StatusCode(500);
```

We start with the `HttpPut` attribute supply a template that is actually identical to the HTTP GET. As you will notice we are taking on the `{id}` for the order that is being updated. The `FromRoute` attribute provides the id argument.

The `FromBody` attribute is what will deserialize the HTTP PUT request body as our C# `Order` instance into the `order` parameter. We express our operation as the invocation to the order services’ `UpdateOrderAsync` function, passing along the `id` and `order`. Finally, based on whether we are able to successfully update the order – we return either an HTTP status code of 200 or 500 for failures to update.

The return HTTP status code of *301, Moved Permanently* should also be a consideration. If we were to add some additional logic to our underlying order service – we could check the given “id” against the order attempting to be updated. If the “id” doesn’t correspond to the give order, it might be applicable to return a `RedirectPermanent` - 301 passing in the new URL for where the order can be found.

HTTP DELETE Request

The last operation on our agenda is the *delete* operation and this is exposed via an action that handles the HTTP DELETE request. There is a lot of debate about whether an HTTP DELETE should be idempotent or not. I lean towards it not being idempotent as the initial request actually deletes the resource and subsequent requests would actually return an HTTP status code of *204, No Content*.

From the perspective of the route template, we look to REST for inspiration and follow its suggested patterns.

The HTTP DELETE verb is similar to the HTTP GET in that we will use the `{id}` as part of the route and invoke the delete call on the collection of orders. This will delete the order for the given `id`.

```
[
  HttpDelete(“api/orders/{id}”)
]
public async Task<IActionResult> Delete([FromRoute] int id)
  => (await _orderService.DeleteOrderAsync(id))
    ? (IActionResult)Ok()
    : NoContent();
```

While it is true that using the `FromQuery` with an HTTP DELETE request is possible, it is unconventional and ill-advised. It is best to stick with the `FromRoute` attribute.

Conclusion:

The ASP.NET Core framework makes authoring *RESTful* Web APIs simple and expressive. The power of the attributes allow your C# code to be decorated in a manner consistent with declarative programming paradigms. The controller actions’ are self-documenting and constraints are easily legible. As a C# developer – reading an action is rather straight-forward and the code itself is elegant.

In conclusion and in accordance with *RESTful* best practices, the following table depicts which ASP.NET Core attributes complement each other the best.

	[FromRoute]	[FromQuery]	[FromBody]	[FromForm]	[FromHeader]
[HttpGet]	**Yes**	**Yes**	No	No	Maybe[2]
[HttpPost]	No	No	**Yes**	**Yes**	No
[HttpPut]	No	No	**Yes**	**Yes**	No
[HttpDelete]	**Yes**	Maybe[1]	No	No	No

1) The `FromQuery` attribute can be used to take an identifier that is used as a HTTP DELETE request argument, but it is not as simple as leveraging the FromRoute attribute instead.

2) The `FromHeader` attribute can be used as an additional parameter as part of an HTTP GET request, however it is not very common – instead use `FromRoute` or `FromQuery` ■

Microsoft® MVP Most Valuable Professional

David Pine

Author

David Pine is a Technical Evangelist working at Centare, with a passion for learning. David is a technical blogger, whose blogs have been featured on asp.net, msdn webdev and msdn dotnet. David is a public speaker, a social coder, an open-source developer advocate, a mentor and a contributor on stackoverflow.com.

Thanks to Daniel Jimenez Garcia for reviewing this article.

Damir Arh

WHAT'S NEW FOR .NET DEVELOPERS

Microsoft has been recently announcing a slew of new development tools and frameworks for .NET Developers. This post gives a brief overview of what was announced and what is currently available.

.NET Standard and .NET Core – What and Why?

Unless you have been living under a rock, you must have heard by now about .NET Standard and .NET Core.

.NET Standard (www.dotnetcurry.com/dotnet/1377/dotnet-standard-2-xaml-standard) is a formal specification of .NET API's that a Framework must implement to be .NET Standard compliant. This specification is important because it establishes a uniformity in the .NET ecosystem.

As of this writing, there are four .NET implementations that are .NET Standard 2 compliant:

- .NET Framework
- .NET Core
- Xamarin and
- Mono.

In the near future, Universal Windows Platform (UWP) vNext will also implement the .NET Standard spec. The .NET Standard makes it easy to write .NET code that can be shared across all these .NET implementations. So as a developer, if your write a .NET Standard Class Library, you can be rest assured that it will work on all the four .NET implementations listed above, as well as the new ones implemented in the future.

.NET Core

In November 2016, .NET Core 1.1 was released at the Connect() event. This was the first new minor version of .NET Core - a cross-platform, open source, and modular .NET platform for creating modern .NET applications. Unlike the LTS (Long Term Support) version 1.0, .NET Core 1.1 was categorized as a "Current" release.

While LTS releases have a 3-year support period from the original release, or 1 year from the next LTS release; the "Current" releases are only eligible for support as long as their parent LTS release support period or 3 months after the next "Current" release, whichever comes first.

The two versions can safely be installed side-by-side, so that individual applications on the same machine can use either one.

The focus of this release was performance.

The big news was that ASP.NET Core is now included in the TechEmpower web framework benchmarks, and is the fastest mainstream full stack framework in the Plaintext category.

Several new features were added to ASP.NET Core 1.1 as well:

- With MVC filters, you can apply middleware only to specific controllers or actions.
- Two new middleware were included: URL rewriting and response caching.
- Web Listener Server for Windows is available as an alternative to Kestrel. With it, you can take advantage of Windows specific features, such as Windows authentication, HTTP/2 over TLS, and others.

- Compiling views during the publish process is supported.

Editorial Note: *You may want to read http://www.dotnetcurry.com/aspnet/1329/aspnet-core-11-what-is-new to learn more on ASP.NET Core 1.1.*

Entity Framework Core 1.1 has built-in support for connection resiliency and SQL Server memory-optimized tables. Thanks to improved LINQ translation, the database engines can successfully execute more queries than before. New APIs were added for explicit loading, mapping to fields and a few other features.

Editorial Note: *To create an application on Entity Framework Core, read this tutorial http://www.dotnetcurry.com/entityframework/1347/entity-framework-ef-core-tutorial*

Final version of MSBuild tooling for .NET Core was released in March 2017 as part of Visual Studio 2017. This means that standard `.csproj` project files can be used instead of `project.json`. To better align the two formats, MSBuild now supports wildcard-based inclusion of files for folder based projects, and direct NuGet package references. A wizard for migrating `project.json` projects to `.csproj` is available as well.

.NET Core 2.0

In May 2017, .NET Core 2.0 Preview was released at Build 2017 event. Three months later, on August 14th, .NET Core 2.0.0 final version was made available.

The main focus of this first new major version of .NET Core is increased source level compatibility with the full .NET framework. To achieve that, the team added over 20000 APIs from .NET framework, which were missing in .NET Core 1.1.

At the same time it introduced .NET Standard 2.0, which will be implemented by both .NET Framework and .NET Core 2.0 and will allow compatible .NET framework libraries to be used directly from .NET Core 2.0. You can read more about .NET Standard 2.0 in my article about it for Dot Net Curry (DNC) magazine.

.NET Core 2.0 is also accompanied by Entity Framework Core 2.0 and ASP.NET Core 2.0.

Entity Framework Core 2.0 is mostly about improvements to LINQ implementation and reduced functionality gap between Entity Framework 6 and Entity Framework Core. ASP.NET Core 2.0 is reintroducing Razor pages and features simplified startup code.

Support for C# and F#

Both C# and F# are fully supported for .NET Core 2.0 development. The same project templates are available for both languages, however Visual Studio 2017 is not yet fully compatible with .NET Core development in F#. While projects can be built and debugged, IntelliSense does not yet work correctly for F# and will be fixed in one of the future Visual Studio updates.

For the time being, Visual Studio Code (http://www.dotnetcurry.com/visualstudio/1340/visual-studio-code-tutorial) is a better alternative – Ionide-FSharp extension has full IntelliSense support, while the C# extension provides the .NET Core debugger.

Visual Basic support is also being added to .NET Core: it can be used for developing .NET Core based class libraries and console applications.

Visual Studio 2017

In March Visual Studio 2017 was released at a dedicated launch event. The new version comes with many productivity improvements:

- better filtering and default suggestions in IntelliSense dialogs,
- easier navigation with redesigned *Find All References* and *Go To* functionalities,

- new *Exception Helper* dialog with more information and support for filtering exceptions by the assembly they originate from,

- quick reattaching with *Reattach to Process* command and filtering by process name in the *Attach to Process* dialog,

- continuous running of tests and coverage information for individual code lines with Live Unit Testing (only available in Enterprise edition),

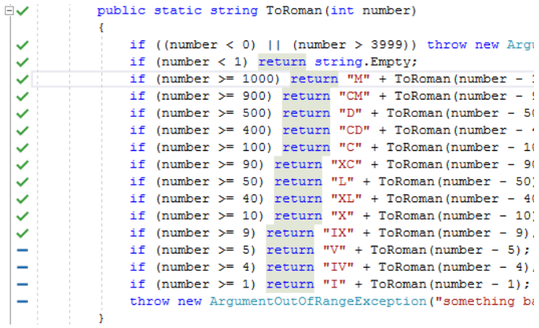


Figure 1: Live Unit Testing in action

- additional code refactorings,
- configurable code style, and more.

Better performance is also a high priority: Visual Studio 2017 starts faster, has shorter load times for solutions, and a faster, more componentized installer. To further speed-up initial loading of large solutions, lightweight solution loading can be enabled for them, which defers the full loading of projects to a later time when they are first needed.

Editorial Note: *You can read more about VS 2017 in our tutorials at http://www.dotnetcurry.com/tutorials/visualstudio*

Extensions can also largely affect the overall responsiveness of Visual Studio. The new performance-monitoring feature will now warn the user about extensions which slow down Visual Studio.
Other improvements to extensions include:

- batched installs and updates,
- built-in roaming extension manager, and
- support for extensions to detect missing dependencies and trigger their installation in Visual Studio.

A few other noteworthy features in Visual Studio 2017 are:

- New versions of C# and Visual Basic provide many new language features, as described in my previous article on C# 7.0 for Dot Net Curry (DNC) magazine.

- A new Service Capabilities page is now the centralized location for adding cloud service dependencies to a project.

- You can record tests for mobile applications using the new mobile test recorder and upload them directly

to Xamarin Test Cloud.

- Visual Studio Tools for Unity are now available as a part of a separate Unity workload in the Visual Studio installer. Among other improvements, the code editor now features full syntax coloring and IntelliSense support for Unity event functions.

With Visual Studio 2017 the release rhythm of updates has also changed. Since the initial release in March, three updates (15.1, 15.2 and 15.3) have already been released, bringing several improvements to Visual Studio, most notably:

- support for .NET Core 2.0,
- Windows 10 Creators Update SDK for UWP applications,
- Redgate Data Tools in Visual Studio Enterprise,
- updated TypeScript support, and
- a new Data Science and analytics workload.

C# 7.x and 8.0

With the release of C# 7.0 as part of Visual Studio 2017, the planned release cadence for new language versions has changed. Between the major versions, several new minor versions will be released, which will allow smaller features to be released sooner, not having to wait for a new major language version to be ready.

According to this new policy, Visual Studio 2017 update 15.3 introduces C# 7.1 with several new features: asynchronous Main method, default literal, inferred tuple names and pattern matching with generics. Other small features are already planned for C# 7.2, such as read-only by-reference argument passing and named arguments in non-trailing positions.

The next major version, C# 8.0, is currently planned to support safe nullable reference types and default interface methods, which promise to bring trait-like behavior to C#.

Visual Studio for Mac

The Visual Studio product family got a new member: Visual Studio for Mac.

The application is an evolved version of Xamarin Studio, more closely aligned to the Visual Studio experience on Windows. It supports development of Xamarin based mobile applications, and their cloud backends in .NET Core, using either C# or F#. It features the same Roslyn powered compiler, IntelliSense code completion and refactoring experience as Visual Studio. The same MSBuild project format allows for seamless sharing of projects between Visual Studio on Windows and Mac.
There's also a preview of the next release available for download. It adds support for Unity development, publishing ASP.NET Core applications to Docker and development of Azure Functions.

Mobile Development with Xamarin

The Xamarin team recently released stable versions of several new tools.
Xamarin Inspector is an extension for Visual Studio and Xamarin Studio, which can attach to a running iOS, Android, Mac or WPF application. It provides a structured look into the current view of the attached application, without pausing its execution with a debugger. You can even modify its properties via a graphical user interface or by executing custom C# code in the application context.

Xamarin Workbooks is an editor and viewer for interactive documents, consisting of rich formatted documentation with live runnable code. This makes it very useful for creating and consuming learning materials, guides and teaching aids. A large selection of workbooks on different Microsoft technologies is already available for download.

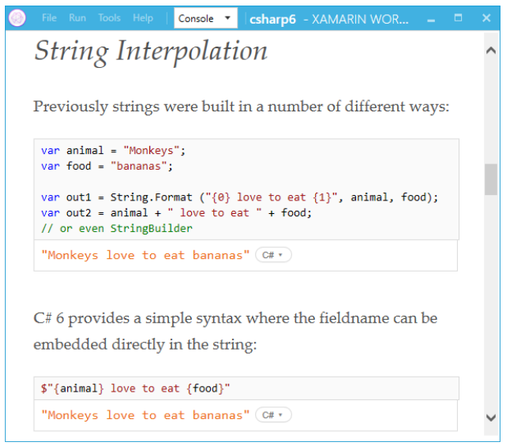


Figure 2: C# 6 workbook in Xamarin Workbooks

Remoted iOS Simulator for Windows provides a convenient way for testing and debugging iOS applications from Visual Studio on Windows. Although the simulator still requires a connected Mac machine where the simulator is actually running, you can fully interact with it directly from Windows. It even provides full touchscreen support, including Apple Pencil simulation using a stylus – that is if you have a touchscreen connected to your Windows machine. The simulator does not support Apple Watch devices yet.

Xamarin Profiler is everything you would expect from a profiler for Xamarin based applications. Its three main views allow tracking of memory allocations and footprint, performance tracking with method execution sampling, and tracking of memory cycles that cannot be released and cause memory leaks.
Remoted iOS Simulator and Xamarin Profiler are only available to owners of Visual Studio Enterprise license.

Editorial Note: *To learn more about Xamarin, visit our tutorials at http://www.dotnetcurry.com/tutorials/xamarin*

At Build 2017 event, another new Xamarin tool was announced to be available in preview: Xamarin Live Player. It comes with an accompanying mobile application for iOS and Android. With some limitations, it allows direct deployment of Xamarin applications from the IDE to your device without having to create and deploy the full package for every change.

Another large Xamarin related product in preview is Visual Studio Mobile Center. It is the successor to HockeyApp and Xamarin Test Cloud, an integrated cloud service for the complete lifecycle of your mobile applications, providing a wide spectrum of functionalities:

- continuous integration and release management,
- automated testing on devices using Xamarin.UITest, Calabash, Appium, XCUITest or Espresso,
- application distribution to your test users and enterprise users,
- crash reporting and usage metrics,
- backend services for authentication, data storage and push notifications, with more coming in the future.

It currently supports UWP, iOS and Android applications written in C#, Objective-C, Swift, Java, Xamarin or Reactive Native. Support for Cordova applications has been announced.

Direct deployment to Google Play store is already available, support for App Store, Windows Store and Intune has also been announced.

Visual Studio Tools for Tizen - Samsung's Linux based OS

Tizen is Samsung's Linux-based open source operating system, used in over 50 million of their devices: smart TVs, wearables, smartphones and IoT devices. At Connect() in November 2016, Samsung announced its intention on collaborating with Microsoft to bring .NET Core to ARM devices, and add Tizen support to Xamarin.Forms.

Since then Samsung has released four preview versions of Visual Studio Tools for Tizen. They provide an alternative to existing C and HTML 5 based programming models for Tizen.

Mobile and TV application development is already supported. Support for other types of devices will follow in the future. All the development is done in C# with .NET Core (2.0 Preview is already supported), Xamarin. Forms for the user interface, and wrappers for many native APIs to access device specific functionalities - everything with full IntelliSense support.

Debugging takes advantage of device emulators, which are included in the download.

The official release is planned for the end of 2017 as part of Tizen 4.0, when the .NET Core runtime will also ship to devices.

Conclusion:

Microsoft continues to offer a great set of benefits and tools for any developer, any app, and any platform.

Since their inception, Microsoft has been using Connect() and Build events to announce many important development related releases. The last incarnation of both events was no exception. From Visual Studio 2017 and Visual Studio for Mac to new Xamarin tooling and new .NET Core versions, there was something of interest for any developer already working with Microsoft tools or technologies. At the same time, as Microsoft expands its focus from Windows to other platforms, these announcements have become increasingly interesting to developers who are not in Microsoft's ecosystem yet.

The new Microsoft is showing that it isn't afraid of making big bets, and delivering on them. Wake up and Code!

Microsoft MVP Most Valuable Professional

Damir Arh
Author

Damir Arh has many years of experience with Microsoft development tools; both in complex enterprise software projects and modern cross-platform mobile applications. In his drive towards better development processes, he is a proponent of test driven development, continuous integration and continuous deployment. He shares his knowledge by speaking at local user groups and conferences, blogging, and answering questions on Stack Overflow. He is an awarded Microsoft MVP for .NET since 2012.

Thanks to Suprotim Agarwal for reviewing this article.

Ravi Kiran

Angular Universal Apps using Node.js

It is needless to say that front-end JavaScript frameworks have taken a leading position in modern web development. The richness these frameworks bring in to the web application is illustrated in most modern websites.

One of the reasons visitors experience this richness is because these sites are highly responsive. A visitor doesn't have to wait for long to see a response after sending a request.

But this richness comes at a cost!

To create this richness, a lot of logic has to be run on the browser using JavaScript. The cost is, the dynamic content rendered on the page using JavaScript cannot be consistently read by all search engines.

While all Search engines (SE's) can scan the content that travels from the server to the client, not all SE's have the capability to scan it when the content is built dynamically at the client's end. So your site cannot be reached through all

SE's if you are rendering the content on the client side.

In addition to this, we don't see readable previews when links of JavaScript applications are shared on social media. Again, this is because the social media sites do not run JavaScript while rendering previews, thus making the pages hard to share.

The solution to this problem is **Server Side Rendering.**

Some of you might think that this approach is taking us back to the days when we had everything rendered from a server. But no, this approach instead enables us to render the initial view of the application from the server and rest of the application, runs on the client side.

It has the following advantages over the traditional SPAs:

- As the initial page is rendered from the server, it can be parsed by the search engines and the content of the page can be reached using search engines
- The user doesn't have to keep looking at the half-baked page, as the complete page with data, is rendered on the client's system
- Social media platforms like Facebook and Twitter can show the preview of the site when the URL of a server rendered application is posted on these platforms.

Editorial Note: *Web crawlers are becoming smarter now-a-days. Google made a statement in October 2015 implying that it can crawl and index dynamic content. Bing can too. However for a consistent SEO experience across all search engines, server side rendering still remains the preferred option.*

Server Side Rendering in Angular

To support server side rendering in Angular, the Angular team created a project named Angular Universal. Angular Universal is a package to add server rendering support to Angular applications. As Angular is a client framework and can't be understood by server platforms, the Angular Universal package provides the required knowledge to the server engine.

As we will see shortly, the universal package can be used to pass the Angular module to the Node.js server¬. Before Angular 4, it was maintained as a separate repository, but it is now moved to the npm submodule @ angular/platform-server.

The support for universal apps was added to Angular CLI in the version 1.3.0.

In this article, we will build an application using the pokémon API to show a list of pokémons and their details. The application would be rendered from the server.

Setting up the environment and application

To follow along the steps shown in this article, the following tools need to be installed on your system:

- **Node.js (version 6 or later):** Can be downloaded from the official site for Node.js and installed
- **npm:** A package manager for Node, it is installed along with Node.js
- **Angular CLI:** It is an npm package created by the Angular team to make the process of creating, testing and building an Angular application easier. It can be installed using the following command:

```
> npm install @angular/cli -g
```

Once these tools are installed, a new Angular application can be generated using the following command:

```
> ng new app-name
```

We will be building an application that consumes the Pokéapi and displays some information about the pokémons. As this is a universal application, let's name it as *pokemon-universal*. Run the following command to create this application:

```
> ng new pokemon-universal
```

This command will take a few minutes to complete.

Once it completes, the command creates a new folder named after the project, creates a structure inside the folder and installs the npm packages. The following screenshot shows the structure in Visual Studio Code:

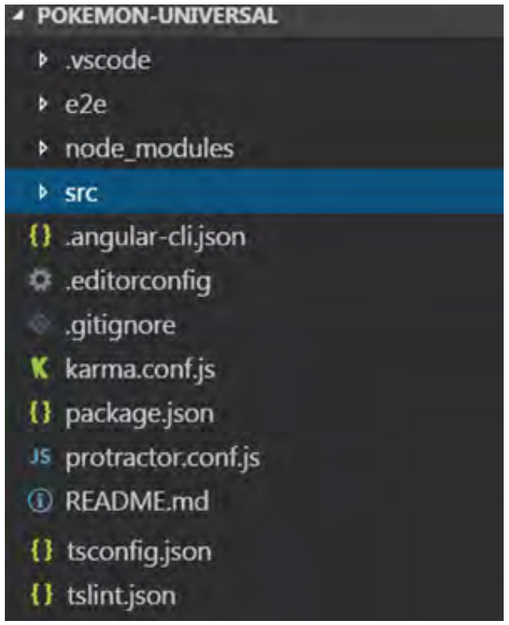


Figure 1 – Folder structure of the project

The following listing explains the vital parts of the project created:

- The src folder will contain the source files of the Angular application. The generated project comes with a few default files

- The file *.angular-cli.json* is the configuration to be used by Angular CLI

- The files *karma.conf.js* and protractor.conf.js contain the settings for the Karma test runner to run unit tests and the settings for protractor to run e2e tests

- The *e2e* folder would contain the end to end tests

- The file *tsconfig.json* contains the configuration to be used to compile TypeScript

- The linting rules to be used during TypeScript compilation are stored in the file *tslint.json*

You can run this application using the following command:

```
> ng serve
```

This command starts a server on the port 4200. Open a browser and change the URL to http://localhost:4200 to browse the application. Now the application is in the default SPA mode.

We need to make a set of changes to be able to run it from the server. Let's do this in the next section.

Installing Packages

Let's make the default application render from server before building the pokémon app.

For this, a couple of packages need to be installed to enable the application for server rendering. The following list describes them:

1. @angular/platform-server: This package is created by the Angular team to support server side rendering of Angular applications. This module will be used to render the Angular module from the server side Node.js code

2. express: Express is a Node.js framework for building web applications and APIs. This package will be used to serve the page

The following commands will install these packages. Open a command prompt, move to the folder where the sample app is created and run the following commands there:

```
> npm install @angular/platform-server --save-dev
> npm install express --save
```

Adding Required Files

We need to make a few changes to the Angular application to enable it for server rendering.

The *BrowserModule* imported in the *AppModule* located in the file *app.module.ts* has to be made server compatible by adding *withServerTransition*. Change it as shown in the following snippet:

```
imports: [
  BrowserModule.withServerTransition({ appId: 'my-app' })
]
```

A new module has to be created specifically to run the code from the server. This module would import the main *AppModule* that starts the application and the *ServerModule* exported by the @angular/platform-server package. The new module won't have components and other code blocks of its own.

Add a new file to the *app* folder and name it *app-server.module.ts*. Place the following code inside this file:

```
import { NgModule } from '@angular/core';
import { ServerModule } from '@angular/platform-server';

import { AppModule } from './app.module';
import { AppComponent } from './app.component';

@NgModule({
  imports: [
    AppModule,
    ServerModule
  ],
  declarations: [],
  bootstrap: [AppComponent]
})
export class AppServerModule { }
```

We need a file to serve as the entry point for the application from the server. This file does nothing more than importing the *AppServerModule* created above. Add a new file to the *src* folder and name it *main.server.ts*.

The following snippet shows the code to be added to this file:

```
export { AppServerModule } from ‘./app/app-server.module’;
```

The TypeScript files have to be transpiled differently for server side rendering. For this, a different configuration file is needed for transpilation. Add a new file to the src folder and name it tsconfig.server.json.

Paste the following code in this file:

```
{
  “extends”: “../tsconfig.json”,
  “compilerOptions”: {
    “outDir”: “../out-tsc/app”,
    “baseUrl”: “./”,
    “module”: “commonjs”,
    “types”: []
  },
  “exclude”: [
    “test.ts”,
    “**/*.spec.ts”
  ],
  “angularCompilerOptions”: {
    “entryModule”: “app/app-server.module#AppServerModule”
  }
}
```

You will notice the following differences between this file and the tsconfig.app.json file located in the src folder:

- The module system to be transpiled is set to commonjs
- It has a new section named angularCompilerOptions and it specifies the path of the Angular module to be loaded from the server code

Adding Configuration for Server Rendering

Information about server rendering has to be fed to the Angular CLI.

For this, we need to add a new application entry in the file *.angular-cli.json*. If you open this file, you will see a property named apps assigned with an array that has a single entry. The existing entry is the configuration used for development and deployment of the application in the client-oriented mode. The new entry will be for server-oriented mode.

The following is the configuration entry to be added to the *apps* array:

```
{
  “platform”: “server”,
  “root”: “src”,
  “outDir”: “dist-server”,
  “assets”: [
    “assets”,
    “favicon.ico”
  ],
  “index”: “index.html”,
  “main”: “main.server.ts”,
  “test”: “test.ts”,
  “tsconfig”: “tsconfig.server.json”,
  “testTsconfig”: “tsconfig.spec.json”,
  “prefix”: “app”,
  “styles”: [
    “styles.scss”
  ],
  “scripts”: [],
  “environmentSource”: “environments/environment.ts”,
  “environments”: {
    “dev”: “environments/environment.ts”,
    “prod”: “environments/environment.prod.ts”
  }
}
```

As you can see, most of the configuration in this entry is similar to the first entry, albeit some differences which are as follows:

- The first property *platform* set to server says that this application is going to be executed from the server. The ng build command uses this value to generate server friendly code when the build is executed
- The output path of this application is set to *dist-server*, which means the files created by this build would be placed in the dist-server folder
- It uses the new *tsconfig.server.json* file to *transpile TypeScript*

Setting up the Node.js Server

The last file to be added to the application is the file that starts the node.js server. This file will use the JavaScript file produced by running production build using the server application configured in the *.angular-cli.json* file. It is then applied on the *index.html* page. The resultant *index.html* file would be served by the express engine.

Add a new file to the root of the application and name it *server.js*. Add the following content to this file:

```
let express = require(‘express’);
let path = require(‘path’);
let ngCore = require(‘@angular/core’);
let fs = require(‘fs’);

let app = express();
const PORT = 3000;

// Load zone.js for the server
require(‘zone.js/dist/zone-node’);

// Enabling prod mode
ngCore.enableProdMode();

// Import renderModuleFactory from @angular/platform-server
let renderModuleFactory = require(‘@angular/platform-server’).renderModuleFactory;

// Load the index.html file from the dist folder
```

```
let index = fs.readFileSync('./dist/index.html', 'utf8');

//----------------------------------------------------------------------------
// Import the AOT compiled factory for your AppServerModule
// This import will change with the hash of your built server bundle
let AppServerModuleNgFactory = require('./dist-server/main.bundle').
AppServerModuleNgFactory;
app.engine('html', (_, options, callback) => {
  const opts = { document: index, url: options.req.url };
  // Render to HTML and send it to the callback
  renderModuleFactory(AppServerModuleNgFactory, opts).then(html => callback(null,
html));
});
//----------------------------------------------------------------------------

app.set('view engine', 'html');
app.set('views', 'dist')
app.get('*.*', express.static(path.join(__dirname, '.', 'dist')));

// Respond with the content read from index for all requests
app.get('*', (req, res) => {
  res.render('index', { req });
});

app.listen(PORT, () => {
  console.log(`listening on http://localhost:${PORT}!`);
});
```

The most important parts of the code in the snippet have inline comments explaining them. As you see, the server code uses the static files from the *dist* folder to serve the client application. The most vital part of this file is the snippet shown in the following screenshot:

```
21  // Import the AOT compiled factory for your AppServerModule
22  // This import will change with the hash of your built server bundle
23  let AppServerModuleNgFactory = require('./dist-server/main.bundle').AppServerModuleNgFactory;
24  app.engine('html', (_, options, callback) => {
25      const opts = { document: index, url: options.req.url };
26      // Render to HTML and send it to the callback
27      renderModuleFactory(AppServerModuleNgFactory, opts).then(html => callback(null, html));
28  });
```

Figure 2 – Code snippet loading the Angular module in the server

Statement 23 gets object of the Angular module built for server rendering. This module is passed to the *renderModuleFactory* method along with the options. The options contain the HTML content to be rendered and the URL to be served by the Angular application. The URL is useful in applications with routes.

The HTML obtained as a result of this operation is passed to the callback of the HTML engine, so that it can be rendered on the page.

Running the Default Application in Server Mode

Now we are good to build and run the application.

Run the following commands in the given sequence. The first command builds the client application, second command builds the code to be rendered from the server and the third command starts the Node.js server.

```
> ng build
> ng build --app 1 --prod --output-hashing none
> node server.js
```

Open your favorite browser and change the URL to http://localhost:3000. You will see the same application that we saw earlier. But the difference is, now the components are compiled on the server and the result is sent to the browser. The following screenshot taken on Chrome dev tools shows the content served in response to the index file:

Figure 3 – Content served for index.html from server

***Note**: Some of these instructions may change in the future versions of angular-cli or platform-server. You can refer to the instructions on the official wiki of angular-cli if these steps don't produce the desired result.*

Building the Pokémon Explorer App

Now that we saw how the default application works when rendered from the server, let's modify the sample to add two routes and see how they work when rendered from server.

The final application will have two routes.

One to show a list of pokémons and the second one to show the details of a pokémon selected on the first page. First, we need to add a service to fetch the data and create a model to represent the structure of a pokémon.

We need bootstrap in the application we are going to build. We need to install bootstrap from npm and add it to *.angular-cli.json* file to include it in the bundle. Run the following command to install the package:

```
> npm install bootstrap --save
```

Modify the *styles* property in both the applications configured in *.angular-cli.json* as following:

```
"styles": [
  "styles.css",
  "../node_modules/bootstrap/dist/css/bootstrap.css"
]
```

Fetching Pokémon Data

The following snippet shows the model classes required for the pokémon explorer. Add a new file to the *app* folder, name it *pokemon.ts* and add the following code to it:

```
export class Pokemon {
  name: string;
  id: number;
  types = [];
  stats = [];
  sprites: Sprite[] = [];

  get imageUrl() {
    return `https://rawgit.com/PokeAPI/sprites/master/sprites/pokemon/${this.id}.
png`;
  }
}

export class Sprite {
  name: string;
  imagePath: string;
}
```

The service will make use of the above models to serve the data to the components. To add the service, you can run the following Angular CLI command:

```
> ng g s pokemon.service
```

Once the file is generated, add the following code to it:

```
import { Injectable } from '@angular/core';
import { Http } from '@angular/http';
import 'rxjs/add/operator/toPromise'

import { Pokemon } from './pokemon';

@Injectable()
export class PokemonService {

 private baseUrl: string = 'https://pokeapi.co/api/v2';

 constructor(private http: Http) { }

 listPokemons() {
  return this.http.get(`${this.baseUrl}/pokedex/1/`)
   .toPromise()
   .then((res: any) => {
    let pokemons: Pokemon[] = [];
    let reducedPokemonEntries = JSON.parse(res._body).pokemon_entries.splice(0,
50);

    reducedPokemonEntries.forEach((entry) => {
     let pokemon = new Pokemon();
     pokemon.name = entry.pokemon_species.name;
     pokemon.id = entry.entry_number;

     pokemons.push(pokemon);
    });
    return pokemons;
  });
 }

 getDetails(id: number) {
  return this.http.get(`${this.baseUrl}/pokemon/${id}/`)
   .toPromise()
   .then((res: any) => {
    let response = JSON.parse(res._body);
    let pokemon = new Pokemon();
    pokemon.name = response.name;
    pokemon.id = response.id;

    response.types.forEach((type) => {
     pokemon.types.push(type.type.name);
    });

    response.stats.forEach((stat) => {
     pokemon.stats.push({
      name: stat.stat.name,
      value: stat.base_stat
     });
    });

    for (let sprite in response.sprites) {
     if (response.sprites[sprite]) {
      pokemon.sprites.push({
      name: sprite,
      imagePath: response.sprites[sprite]
     });
    }

    return pokemon;

   });
  }
}
```

The service has two methods. The first method *listPokemons* gets a list of pokémons by querying the pokedex API and takes the first fifty pokémons. This is done to show lesser amount of data on the page and keep the demo simple. If you want to see more number of pokémons, you can modify the logic.

The second method *getDetails* receives an id of the pokémon and gets the details of it by querying the pokémon API. To call the pokémon REST APIs, the service uses *HttpClient*. It is a new service added to the @angular/common package and it has a simplified API to invoke and intercept the HTTP based endpoints.

The service has to be registered as a provider in the module. The following snippet shows the modified *AppModule*:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { HttpModule } from '@angular/http';


import { AppComponent } from './app.component';

import { PokemonService } from './pokemon.service';
```

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule.withServerTransition({ appId: 'my-app' }),
HttpModule

  ],
  providers: [PokemonService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Building the Components

Let's add the components required for the application.

We need two components to show the list of pokémons and to show the details of a pokémon. These components would be configured to render on different routes. Let's get the components generated, then we will modify them to display the data we need to show.

```
> ng g c pokemon-list -m app.module
> ng g c pokemon-details -m app.module
```

Note: *Notice the –m option added to the above commands. This option tells Angular CLI to register the generated component in the specified module. We don't need to specify this option in most of the cases. But here we need it because we have multiple modules created at the root of the application.*

As the name says, the *pokemon-list* component simply lists the pokémons. It calls the *getList* method of the *PokemonService* and displays the results in the form of boxes. Open the file *pokemon-list.component.ts* and replace the code of this file with the following:

```
import { Component, OnInit } from '@angular/core';
import { PokemonService } from '../pokemon.service';
import { Pokemon } from '../pokemon';

@Component({
  selector: 'app-pokemon-list',
  templateUrl: './pokemon-list.component.html',
  styleUrls: ['./pokemon-list.component.css']
})
export class PokemonListComponent implements OnInit {
  pokemonList: Pokemon[];

  constructor(private pokemonService: PokemonService) { }

  ngOnInit() {
    this.pokemonService.listPokemons()
      .then(pokemons => {
        this.pokemonList = pokemons;
      });
  }
}
```

The template of this component has to be modified to show the list. The widget showing each pokémon will have a link to navigate to the details page. Replace the content in the file *pokemon-list.component*.html with the following:

```
<div *ngFor="let pokemon of pokemonList" class="col-md-3 col-md-offset-1 text-
center box">
 <div class="row">
  <div class="col-md-12">
   <img [src]="pokemon.imageUrl" height="150" width="150" />
  </div>
  <div class="row">
   <div class="col-md-12 link">
    <a [routerLink]="['/details',pokemon.id]">{{ pokemon.name | titlecase }}</a>
   </div>
  </div>
 </div>
</div>
```

The *pokemon-details* component receives id of the pokémon whose details have to be displayed and it calls the *getDetails* method of the *PokemonService* using received id to fetch the details.

The following snippet shows this code. Replace the code in the file *pokemon-details.component.ts* with the following:

```
import { Component, OnInit } from '@angular/core';
import { Router, ActivatedRoute, ParamMap } from '@angular/router';
import { PokemonService } from '../pokemon.service';
import { Pokemon } from '../pokemon';

@Component({
  selector: 'app-pokemon-details',
  templateUrl: './pokemon-details.component.html',
  styleUrls: ['./pokemon-details.component.css']
})
export class PokemonDetailsComponent implements OnInit {
  id: number;
  pokemon: Pokemon;

  constructor(private route: ActivatedRoute,
    private router: Router,
    private pokemonService: PokemonService) { }

  ngOnInit() {
    this.route.paramMap.subscribe((params) => {
      this.id = parseInt(params.get('id'));
      this.pokemonService.getDetails(this.id)
        .then((details) => {
          this.pokemon = details;
        });
    });
  }
}
```

This component has to display the details of the pokémon like name, types, statistics and images of the sprites. The following snippet shows the template, place this code in *pokemon-details.template.html*:

```
<div *ngIf=”pokemon” class=”details-container”>
  <img [src]=”pokemon.imageUrl”>
  <div>{{pokemon.name | titlecase}}</div>
  <h4>Types:</h4>
  <ul>
    <li *ngFor=”let type of pokemon.types”>
      {{ type }}
    </li>
  </ul>

  <h4>Stats:</h4>
  <ul>
    <li *ngFor=”let stat of pokemon.stats”>
      {{ stat.name }}: {{ stat.value }}
    </li>
  </ul>

  <h4>Sprites:</h4>
  <div *ngFor=”let sprite of pokemon.sprites” class=”col-md-3”>
    <img [src]=”sprite.imagePath” />
    <br>
    <span>{{sprite.name}}</span>
  </div>
</div>
```

Adding Routes

Now that the components are ready, let’s add the routes and complete the application.

Add a new file in the *src* folder and name it *app.routes.ts*. As we have been discussing till now, we need to add two routes in the application. One to show the list of pokémons and the other to show details of a pokémon.

The following snippet shows the code. Add the following code to this file:

```
import { Routes, RouterModule } from ‘@angular/router’;
import { PokemonListComponent } from ‘./pokemon-list/pokemon-list.component’;
import { PokemonDetailsComponent } from ‘./pokemon-details/pokemon-details.
component’;

let routes: Routes = [
  {
    path: ‘’,
    component: PokemonListComponent
  },
  {
    path: ‘details/:id’,
    component: PokemonDetailsComponent
  }
];

const routesModule = RouterModule.forRoot(routes);
export { routesModule };
```

This has to be added to the application module to make the routing work. We need to import the *routesModule* exported from the above file and add it to the *imports* array of the module.

The following snippet shows the modified module file:

```
import { BrowserModule } from ‘@angular/platform-browser’;
import { NgModule } from ‘@angular/core’;
import { HttpModule } from ‘@angular/http’;

import { routesModule } from ‘./app.routes’;

import { AppComponent } from ‘./app.component’;
import { PokemonListComponent } from ‘./pokemon-list/pokemon-list.component’;

import { PokemonService } from ‘./pokemon.service’;
import { PokemonDetailsComponent } from ‘./pokemon-details/pokemon-details.
component’;

@NgModule({
  declarations: [
    AppComponent,
    PokemonListComponent,
    PokemonDetailsComponent
  ],
  imports: [
    BrowserModule.withServerTransition({ appId: ‘my-app’ }),
HttpModule,routesModule
  ],
  providers: [
    PokemonService
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

The last change we need to make is, modify the template of AppComponent to load the routes. Open the file app.component.html and replace the content of this file with the following code:

```
<div class=”container”>
 <div style=”text-align:center”>
  <h1>
   Explore the Pokemons!
  </h1>
 </div>
 <router-outlet></router-outlet>
</div>
```

Now run the following commands to build and run the application:

```
> ng build
> ng build --app 1 --prod --output-hashing none
> node server.js
```

You will see the pokemon images displayed on the page as shown in Figure 4:

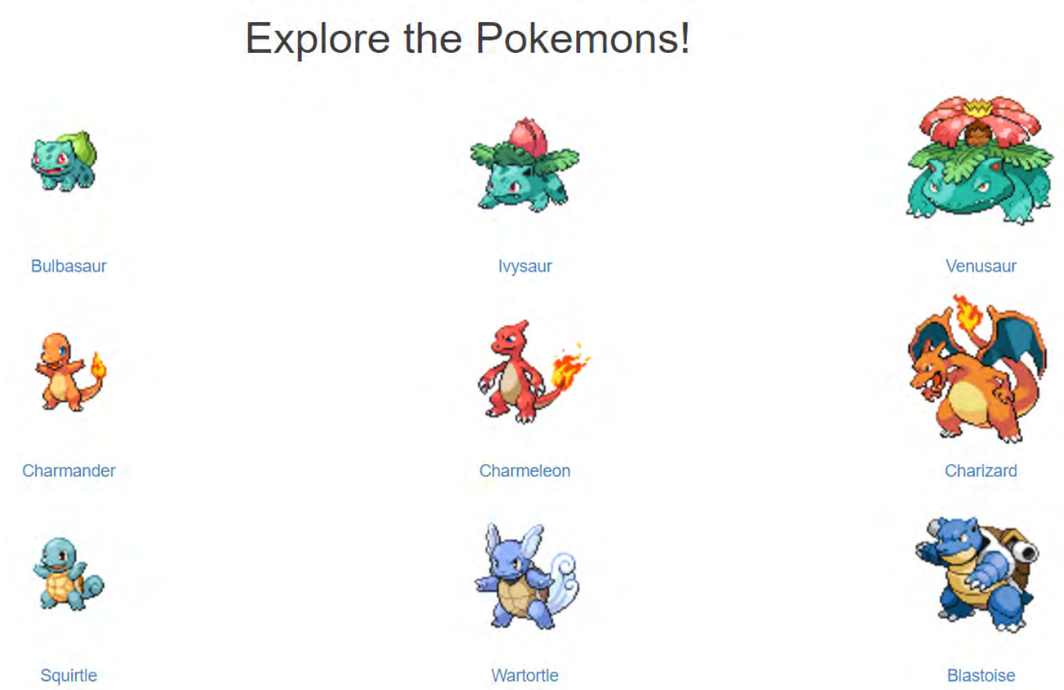


Figure – 4 Pokemon list

Move to the details page by clicking on one of the links and refresh the details page. You will see that the details page is rendered from the server. The following screenshot shows the HTML received from the server in response to the request made:

Figure – 5 HTML of pokemon list from server

Switch between the list and details pages and randomly refresh any of the pages. You will see that the content of the first load comes from the server. This model enables server rendering on all the pages in the application and hence the entire application would have a consistent SEO experience.

Note: *You will notice a slight flicker in the page when it is rendered from the server. The flicker is because the app gets bootstrapped from the client side as well after the rendering completes from the server.*

Conclusion

The wide usage of JavaScript frameworks has made it necessary that we provide consistent SEO experience even for those search engines that cannot scan and index dynamically generated content. Server side rendering is a big value add to business driven sites as they will enjoy the benefits of both being rich and searchable.

As we saw in this article, Angular has a very good support for server side rendering and it is now integrated with Angular-CLI as well. We should definitely use this feature to make our sites SEO consistent! ■

Download the entire source code from GitHub at
bit.ly/dncm32-angular-pokemon

Microsoft® Most Valuable Professional

Ravi Kiran
Author

Rabi Kiran (a.k.a. Ravi Kiran) is a developer working on Microsoft Technologies at Hyderabad. These days, he is spending his time on JavaScript frameworks like AngularJS, latest updates to JavaScript in ES6 and ES7, Web Components, Node.js and also on several Microsoft technologies including ASP.NET 5, SignalR and C#. He is an active blogger, an author at SitePoint and at DotNetCurry. He is rewarded with Microsoft MVP (ASP.NET/IIS) and DZone MVB awards for his contribution to the community.

Thanks to Mahesh Sabnis and Suprotim Agarwal for reviewing this article.

A MAGAZINE FOR .NET AND JAVASCRIPT DEVS

- ☑ AGILE
- ☑ ASP.NET
- ☑ MVC, WEB API
- ☑ ANGULAR.JS
- ☑ NODE.JS
- ☑ AZURE
- ☑ VISUAL STUDIO
- ☑ .NET
- ☑ C#, WPF

We've got it all!

100K PLUS READERS

280 PLUS AWESOME ARTICLES

31 EDITIONS

FREE SUBSCRIPTION USING YOUR EMAIL

EVERY ISSUE DELIVERED

RIGHT TO YOUR INBOX

NO SPAM POLICY

SUBSCRIBE TODAY!

.NET & JavaScript Tools

Shorten your Development time with this wide range of software and tools

CLICK HERE