# We have Liftoff: .NET Core 2.0 and ASP.NET Core 2.0 have Arrived

Many of us have been patiently waiting through the long and winding road that has been the inception of the .NET Core and ASP.NET Core platforms. After a very rocky set of 1.x releases, version 2.0 of the new .NET frameworks and tooling finally arrived a few months back. You know the saying: "Don't use any 1.x product from Microsoft," and this is probably truer than ever with .NET Core and ASP.NET Core. The initial releases, although technically functional and powerful, were largely under-featured and sported very inconsistent and ever-changing tooling. Using the 1.x (and pre-release) versions involved quite a bit of pain and struggle just to keep up with all the developments along the way.

### .NET Core 2.0 and ASP.NET Core 2.0: The Promised Land?

Version 2.0 of .NET Standard, .NET Core, and ASP.NET Core **improve the situation considerably** by significantly refactoring the core feature set of .NET Core, without compromising all of the major improvements that the new framework brought in version 1.

The brunt of the changes involves bringing back APIs that existed in the full .NET framework and make .NET Core 2.0 and .NET Standard 2.0 more backward compatible. It's now vastly easier to move existing full-framework code to .NET Core/Standard 2.0. It's hard to understate how important that step is, as 1.x was hobbled by missing APIs and sub-features that made it difficult to move existing full-framework code and libraries forward to .NET Core/Standard. Bringing the API breadth back to close compatibility with the full framework resets expectations of what’s considered a functional set of .NET features, the kind that most of us have come to expect from .NET as a platform.

These subtle but significant improvements make the overall experience of the 2.0 .NET Core and ASP.NET Core releases much more approachable, especially when coming from previous versions of .NET. More importantly, third parties can now more easily move their components to .NET Core so that the support eco-system for .NET Core applications doesn't feel like a backwater, as it did during the v1 days.

These changes are as welcome as they were necessary and my experience with the 2.0 wave of tools so far has been very positive. I've been able to move two of my most popular libraries to .NET Core 2.0 with relatively little effort, something that would have been unthinkable with the v1 versions. I’ve also finally started building a number of internal Web applications with ASP.NET Core 2.0 and the overall feature breadth is pretty close to full framework, minus the obvious Windows-specific feature set.

ASP.NET Core 2.0 also has many welcome improvements, including simplified configuration that provides sensible defaults, so that you don't have to write the same obvious startup code over and over. There are also many new small enhancements as well as a major new feature in the form of RazorPages that bring controller-less Razor pages to ASP.NET Core.

Overall, 2.0 is a massive upgrade in functionality bringing back features that realistically should have been there from the very beginning.

But it's not all unicorns and rainbows. There are still many issues that need to be addressed moving forward. First and foremost is that the new SDK-style project tooling is still struggling with many small missing features and performance problems. Visual Studio, in particular, seems to have taken a big step backward in stability in the .NET Core release cycles, but the good news is that Microsoft is finally taking tooling performance and stability seriously. The last few Visual Studio point release updates, even since the release of .NET Core/ASP.NET Core 2.0, have made big strides in addressing stability and performance. But there’s still some ways to go.

### The Good Outweighs the Bad

Overall, the improvements in this 2.0 release vastly outweigh the relatively few—if not insignificant—problems, that still need to be addressed. I’ve been waiting for a long time for a .NET Core/Standard version that looks production-ready. Here’s where I stand:.NET Core 2.0 and ASP.NET 2.0 is my demarcation line, my line in the sand, where I get my butt off my hands and finally jump in. It’s been a long wait to get to this point.

.NET Core 2.0 and ASP.NET 2.0 is my demarcation line, my line in the sand, where I get my butt off my hands and finally jump in.

The 2.0 release feels like a good jumping-in point, to dig in and start building real applications with. This is a feeling that I never had with the v1 releases. In v1, I dabbled and worked through my samples and learned, but with V1 releases, I never felt like I'd get through a project without getting stuck on some missing piece of kit.

For me, v2.0 strikes the right balance of new features, performance, and platform options that I actually want to use, without giving up many of the conveniences that earlier versions of .NET offered. The 2.0 features no longer feel like a compromise between the old and new but a way forward to new features and functionality that’s really useful and easy to work with in ways that you would expect on the .NET platform. Plus, there are many hot and shiny new features in ASP.NET Core. There's lots to like and always has been in ASP.NET Core.

Let’s take a look at some of the big improvements in .NET Core 2.0 and ASP.NET Core 2.0.

### Bringing Back Many .NET APIs

The first and most significant improvement in the 2.0 releases is that .NET Standard and .NET Core 2.0 bring back many of the .NET Framework APIs we've been using since the beginnings of .NET in the full framework, but that weren’t supported initially by .NET Standard and .NET Core v1.

The terms .NET Core and .NET Standard will come up a few times in this article and the concepts of the framework and specification are somewhat important. If you’re new to these terms, please see the sidebar .NET Core and .NET Standard.

.NET Standard 2.0 is a specification, a blueprint of features that have to be implemented by a specific platform. .NET Core 2.0 is a specific platform implementation of that specification.

#### .NET Core 1 Was Too Lean

When .NET Core 1.x came out, it was largely touted as a trimmed down, high-performance version of .NET. As part of that effort, there was a lot of focus on *trimming the fat* and providing only core APIs in .NET Core and .NET Standard. The bulk of the .NET Base Class Library was also broken up into a huge number of small hyper-focused packages.

All of this resulted in a much smaller framework, but unfortunately also brought a few problems:

* Major incompatibilities with classic .NET Framework code (it was hard to port code)
* A huge clutter of hard-to-discover NuGet Packages in projects
* Many usability issues when trying to perform common tasks
* A lot of mental overhead trying to combine all of the right pieces into a whole

With .NET Core 1, many common NET Framework APIs were either not available or they were buried under different API interfaces that were often missing critical functionality. Not only was it hard to find stuff that was under previously well-known APIs, but a lot of functionality that was taken for granted (Reflection, Data APIs, and XML, for example) was refactored down to near un-usability.

#### Bringing Back Full Framework Features

.NET Core 2.0—and more importantly **.NET Standard 2.0**--adds back a ton of functionality that was previously cut from .NET Core/Standard. In v1 it was really difficult to port existing code. The feature footprint with .NET Standard 2.0 and .NET Core 2.0 is drastically improved (~150% of new APIs added over v1.1) and compatibility with existing full framework functionality is preserved for a much larger percentage of code. In real terms, this means that it's now much easier to port existing full framework code to .NET Standard or .NET Core and have it run with no or minor changes.

Case in point: I took one of my **15-year-old** general purpose libraries,--Westwind.Utilities, which contains a ton of varied utility classes that touch a wide variety of .NET features—and re-targeted it to .NET Standard. More than 95% of the library migrated without changes and only a few small features needed some tweaks (encryption, database). A few features had to be cut out (low-level AppDomain management and System.Drawing features). Given that this library was such an unusual hodgepodge of features, more single-focused libraries will fare much better for conversions. If you're not using a few of the APIs that have been cut or only minimally implemented, chances are that porting to .NET Standard will require few or even no changes.

You can read a lot more detail about what’s involved in porting a full framework .NET library to .NET Standard in this blog post: Multi-Targeting and Porting a .NET Library to .NET Core 2.0 (<https://goo.gl/e1ADWe>).

#### Runtimes Are Back

One of the key bullet points that Microsoft touted with .NET Core is that you can run **side by side** installations of .NET Core. You can build an application and ship all the runtime files and dependencies in a local folder, including all the .NET dependencies as part of your application. The benefit of this is that you can much more easily run applications that require different versions of .NET on the same computer. No more trying to sync up and potentially break applications due to global framework updates. Yay, right?

.NET Core 2 brings back the concept of shared system runtimes to bring down the size of deployed applications without compromising application level runtime versioning

#### .NET Core 1: Fragmentation and Deployment Size

Well, you win some and you lose some. With Version 1, the side-effect was that the .NET Core and ASP.NET frameworks were fragmented into a boatload of tiny, very focused NuGet packages that had to be referenced explicitly in every project. These focused packages are a boon to the framework developers as they allow for nice feature isolation and testing, and the ability to rev versions independently for each component.

But the result of all this micro-refactoring was that you had to add every tiny little micro-refactored NuGet Package/Assembly explicitly to each project. Finding the right packages to include was a big usability problem for application and library developers.

Additionally, when you published Web projects, all of those framework files—plus all runtime dependencies—had to be copied to the server, making for a huge payload to send up to a server for publishing even for a small HelloWorld Web application.

#### Meta Packages in 2.0

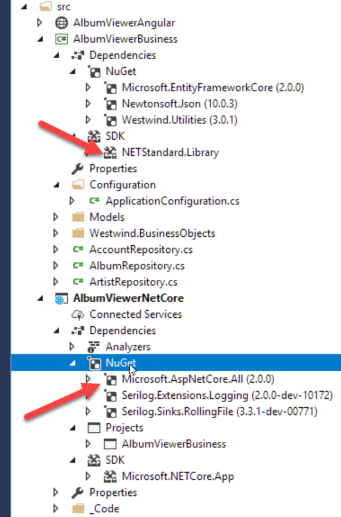
In .NET Core 2.0 and ASP.NET 2.0, this is addressed with system-wide **Framework Meta Packages** that can be referenced by an application. These packages are installed using either the SDK install or a “runtime” installer and can then be referenced from within a project as a single package. When you reference **.NET Core App 2.0** in a project, it automatically includes a reference to the .NET Core App 2.0 meta package. Likewise, if you have a class library project that references **.NET Standard 2.0,** that meta package with all the required .NET Framework libraries is automatically referenced. You don't need to add any of the micro-dependencies to your project. The runtimes reference **everything** in the runtime, so in your code, you only need to apply namespaces but no extra packages or references.

There's also an ASP.NET Core meta package that provides **all of the ASP.NET Core and Entity Framework** references. Each of these meta packages have a very large predefined set of packages that are automatically referenced and available to your project's code.

This means that .NET Core projects no longer pre-load assemblies on startup, so if you don’t use an assembly, it’s never loaded and the runtime JIT compiler ensures that only the code you reference is compiled and loaded into memory. Even though you reference everything and the kitchen sink, your application only loads what it needs at runtime.

In your projects, this means you can reference .NET Standard in a class library project and get references to all the APIs that are part of .NET Standard with a **NetStandard.Library** reference, as shown in **Figure 1**. In applications, you can reference **Microsoft.NETCoreApp,** which is a reference to .**NET Core 2.0.** Here you specify an instance of runtime for the top-level application. For ASP.NET, the **Microsoft.AspNetCore.All** package brings in all ASP.NET and Entity Framework-related references in one simple reference.

**Figure 1** shows an example of a two-project solution that has an ASP.NET Core Web app and a .NET Standard targeted business logic project:



**Figure 1:** Package references are manageable again in V2.0.

Notice that the project references look very clean overall; I only explicitly add references to third-party NuGet packages. All of the system references come in automatically via the single meta package. This is even less cluttered than a full framework project, which still needed some high-level references. Here everything is automatically available for referencing.

This also is nice for tooling that needs to find references (Ctrl-. in VS or Alt-Enter for Resharper). Because **everything** is essentially referenced, Visual Studio or OmniSharp can easily find references and namespaces and inject them into your code as using statements. Nice!

#### Runtimes++

In a way, these meta packages feel like classic .NET runtime installations. Microsoft now provides .NET Core and ASP.NET Core runtimes that are installed from the .NET Download site (<https://www.microsoft.com/net/download>) and can either be installed as only the runtime package or the full .NET SDK that includes all the compilers and command line tools so you can build and manage a project.

You can install multiple runtimes side-by-side and they’re available to many applications to share. This means that the same packages don't have to be installed over and over for each and every application, which makes deployments a heck of a lot leaner than in 1.x applications.

You can still fall back to local packages installed with the application's output folder and override global installed packages, so now you get the best of all worlds. You can:

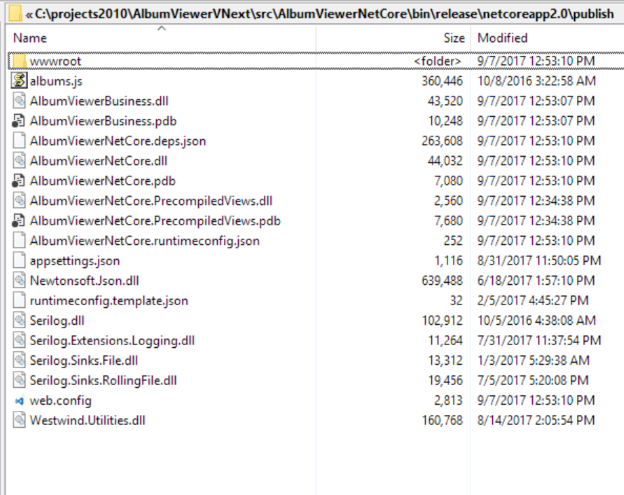
* Run an application with a pre-installed Runtime (default).
* Run an application with a pre-installed Runtime and override packages locally.
* Run an application entirely with locally installed Runtime packages.

In short, you now get to have your cake and eat it too, as you get to choose **exactly** where your runtime files are coming to.

#### Publishing Applications

Using ASP.NET Core 2.0, publishing a Web application to a Web server is much leaner than in prior versions. By default, using shared runtimes, your publish folder contains only **your compiled code** plus any explicit third-party dependencies from NuGet that you added to the project. You’re no longer publishing runtime files to the server so the behavior is once again similar to what you had in a full framework Web application deployment.

**Figure 2** shows the publish folder of the Solution shown in **Figure 1**.



**Figure 2**: Published output contains only your code and your explicit dependencies

This means that publishing your application is no longer a 100mb ordeal, but rather publishes a few megs of the stuff you actually built and use yourself. It's still possible to deploy full runtimes just as you could in v1 releases by explicitly using the **dotnet publish** command line and specifying a specific runtime version to install with, but by default, the publish process uses the pre-installed runtimes.

### .NET SDK Projects

One of the nicest features of 2.0 (initially introduced in 1.1) is the new, SDK style **.csproj** project format. This project format is very lean and easily readable and maintainable, which is quite in contrast to the verbose and cryptic older **.csproj** format.

For example, it's not hard to glean what's going on in the **.csproj** file shown in **Listing 1**.

Notice that explicit file inclusion references are all but gone in the project file. Projects now assume that **all files are included** except those you explicitly exclude, which drastically reduces the content of a project file. The other big benefit is that you can simply drop files in the project folder to become part of the project; you no longer have to add files to a project explicitly.

Compared to the morass that was the old **.csproj** format, this makes it manageable to manually edit the project file as well as improving source control management, as there are much fewer changes than before.

Additionally, the new project format supports multi-targeting to multiple .NET Framework versions. I've talked about porting existing libraries to .NET Standard, and using the new project format, it's quite easy to set up a library to target both .NET 4.5 and .NET Standard, for example.

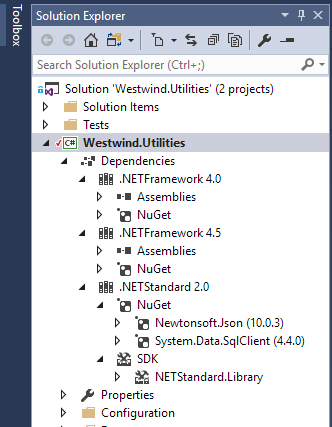
**Listing 2** shows a truncated example of the Westwind.Utilities library that targets both .NET Standard and .NET 4.5 (full triple-target version: <https://goo.gl/x4L3Kd>).

The project defines two framework targets:

<TargetFrameworks>netstandard2.0;net45</TargetFrameworks>

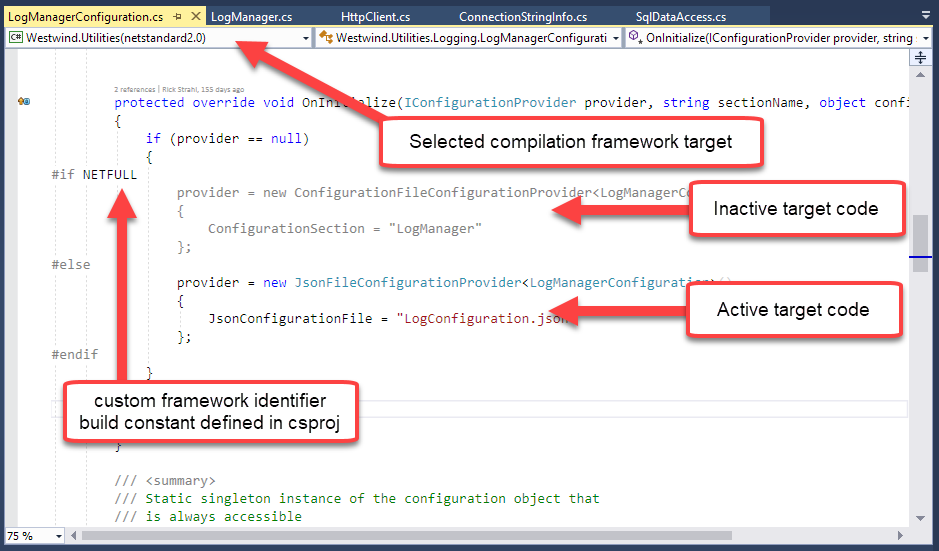
Then it uses conditional target framework filtering to add dependencies. Visual Studio can visualize these dependencies for each target as well, as shown in **Figure 3** (with an additional *net40* target).

.NET SDK projects make it very easy to create a .NET project that targets multiple .NET runtimes and creates a NuGet package.



**Figure 3:** Multiple targets displayed in Visual Studio

Visual Studio 2017.3+ also has a new Target drop down that lets you select which target is currently used to display code and errors in the environment. **Figure 4** shows how Visual Studio visualizes code given a specific runtime target.



**Figure 4:** Visual Studio can detect the active target and highlight code accordingly.

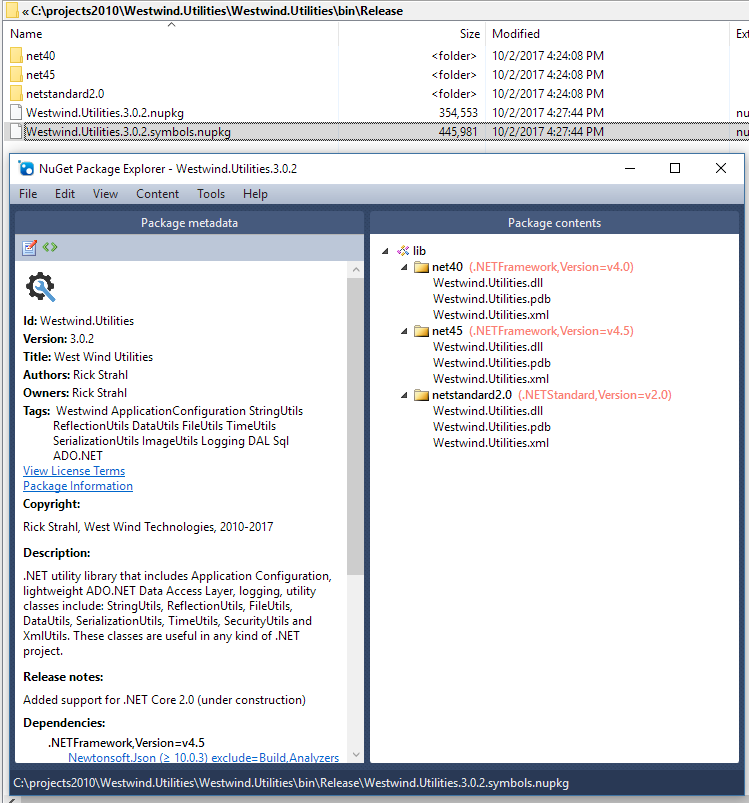
There are other features in Visual Studio that makes it target-aware:

* IntelliSense shows warnings for APIs that don't exist on any given platform.
* Compiler errors now show the target platform for which the error applies.
* Tests using the Test Runner respect the active target platform (VS2017.4+).

When you compile this project, the build system automatically builds output for all targets, which is very nice if you've ever tried to create multi-target projects with the old project system (hint: it sucked!).

The project system can also create a NuGet Package that wraps up all targets into one NuGet package. If you look back at the project file, you'll note that the NuGet Properties are now stored as part of the **.csproj** file.

**Figure 5** shows what the build output from my three-target project looks like.



**Figure 5:** Multi-target projects automagically build for all target platforms and can create a NuGet package.

This is pretty awesome and frankly, something that should have been supported a long time ago in .NET!

### Easier ASP.NET Startup Configuration

Another nice improvement and a sign of growing up is that the ASP.NET Core startup code in 2.0 is a lot more streamlined. There's simply quite a bit less of it.

The **absolute minimal ASP.NET Web application** you can build is just a few lines of code:

public static void Main(string[] args)

{

WebHost.Start(async (context) =>

{

var r = $"Hello, time is: {DateTime.Now}";

await context.Response.WriteAsync(r);

})

.WaitForShutdown();

}

Notice that this code works without any dependencies whatsoever, and yet has access to an **HttpContext** instance—there's no configuration or additional set up required, and the framework now uses a set of common defaults for bootstrapping an application. Hosting options, configuration, logging, and a few other items are automatically set with common defaults, so these features no longer have to be explicitly configured unless you want to change the default behavior.

The code also automatically hooks up hosting for Kestrel and IIS, sets the startup folder, allows for host URL specification and provides basic configuration features - all without any custom set up code. All of these things needed to be configured explicitly previously. Now - all of that is optional. Nice!

To be realistic though, if you build a real application that requires configuration, authentication, custom routing, CORS etc., those things must still be configured and obviously, that will add code. But the point is that ASP.NET Core now has a default configuration that out-of-the-box lets you get stuff done without doing any configuration.

The more common configuration set up looks like this:

public static void Main(string[] args)

{

var host = WebHost.CreateDefaultBuilder(args)

.UseUrls()

.UseStartup<Startup>()

.Build()

.Run()

}

Using a Startup configuration class that handles minimal configuration for an MVC/API application, it looks like this:

public class Startup

{

public void ConfigureServices(

IServiceCollection services)

{

services.AddMvc();

}

public void Configure(IApplicationBuilder app,

IHostingEnvironment env,   
 IConfiguration configuration)

{

app.UseStaticFiles();

app.UseMvcWithDefaultRoute();

}

}

You can then use either **RazorPages** (loose Razor files that can contain code) or standard MVC or API controllers to handle your application logic.

A controller, of course, is just another class you create that optionally inherits from Controller or simply has a Controller postfix:

[Route("api")]

public class HelloController // : Controller

{

[HttpGet("HelloWorld/{name}")]

public object HelloWorld(string name)

{

return new

{

Name = name,

Message = $"Hello World, {name}!",

Time = DateTime.Now

};

}

}

In short, basic configuration for a Web application is now a lot cleaner than in 1.x versions.

One thing that has bugged me in ASP.NET Core is the dichotomy between the ConfigureServices() and Configure() methods. In 1.x, ASP.NET Core seemed to have a personality crisis about where to put configuration code for various components. Some components configured in ConfigureServices() using the AddXXX() methods, others did it in the Configure() method using the UseXXX() methods. In v2.0, Microsoft seems to have moved most configuration behavior into ConfigureServices() using options objects via Action delegates that get called later in the pipeline, so now things like CORS, Authentication, and Logging all use a similar configuration patterns.

For example, in the code in **Listing 3**, DbContext, Authentication, CORS, and Configuration are all configured in the **ConfigureServices()** method.

The Configure() method only enables the behaviors configured in **ConfigureServices()** by using various **.UseXXXX()** methods like **.UseCors("CorsPolicy")**, **.UseAuthentication()**, and **.UseMvc()** etc.

Although this still seems very disjointed, at least the patterns provided by Microsoft keep configuration logic mostly in a single place in **ConfigureServices()**.

Ironically, I've been struggling with this same issue in porting another library, [Westwind.Globalization](https://github.com/RickStrahl/Westwind.Globalization), to .NET Core 2.0 and I needed to decide how to configure my component. I chose to follow the same pattern as Microsoft using **ConfigureServices()** with an Action delegate that handles option configuration:

services.AddWestwindGlobalization(opt =>

{

opt.ResourceAccessMode =

ResourceMode.DbResourceManager;

opt.ConnectionString =config.ConnectionString;

opt.ResourceTableName = "localizations ";

opt.ResxBaseFolder = "~/Properties/";

opt

.ConfigureAuthorizeLocalizationAdministration(

actionContext => return true);

});

implemented as an extension method with an Action delegate:

public static IServiceCollection

AddWestwindGlobalization(

this IServiceCollection services,

Action<DbResConfiguration> setOptionsAction)

{

// add additional services to DI

// configure based on options passed in

}

I'm not a fan of this (convoluted) pattern of indirect referencing and deferred operation, especially given that **ConfigureServices()** seems like an inappropriate place for component configuration when there's a **Configure()** method where I'd expect to be doing any configuring.

But once you understand how Microsoft uses the delegate-option-configuration-pattern, and if you can look past the consistent inconsistency, it’s easy to implement and work with, so I'm not going to rock the boat and do something different in my components.

### IRouterService: Minimal ASP.NET Applications

MVC or API applications are typically built using the MVC framework. As you've seen above, it's a lot easier with 2.0 to get an API application configured and up and running. But MVC has a bit of overhead internally.

IRouterService makes it easy to create simple, routed Delegates that respond to Requests without using MVC.

If you want something even simpler, perhaps for a quick one-off minimal microservice, or you’re a developer who wants to build a custom framework on top of the core ASP.NET middleware pipeline, you can now do that pretty easily by taking advantage of **IRouterService,** which provides access only to the raw HTTP context bits, but no MVC processing framework. You can think of this approach as somewhat similar to an HttpHandler in classic ASP.NET.

**Listing 4** shows another very simple single file, a self-contained ASP.NET application that returns a JSON response off a routed request.

The key is the **IRouterService** passed to the **app.UseRouter(),**which lets you directly map URLs to action delegates that have a request and a response you read from and write to. You can also access **HttpContext** from the **request** or **response** parameters so that all of the context’s intrinsic objects are available. This is obviously a bit lower-level than using MVC/API controllers. There’s no Controller logic available, so you have to handle input deserialization and output generation in your own code. The code above handles its own JSON serialization, for example. With a few simple helper extension methods, you can provide a lot of functionality using just this very simple mechanism. This is useful for publishing simple one-off “handlers.” It can also be a good starting point if you ever want to build your own custom not-MVC Web framework or custom service.

**IRouterService** functionality is primarily for specialized use cases where you need one or more very simple notification request. It’s very a similar use case to where you might employ server-less Web Functions (like Azure Functions or AWS Lambda) for handling simple service callbacks or other one-off operations that have few dependencies.

I've also found **IRouterService** useful for custom route handling that doesn't fall into the application space, but is more of an admin feature. For example, recently I needed to configure an ASP.NET Core app to allow access for Let's Encrypt's domain validation callbacks and I could just use a route handler to handle a special route in the server's Configure() code:

app.UseRouter(r =>

{

r.MapGet(".well-known/acme-challenge/{id}",

async(request, response, routeData) =>

{

var id = routeData.Values["id"] as string;

var file = Path.Combine(env.WebRootPath,

".well-known", "acme-challenge", id);

await response.SendFileAsync(file);

});

});

app.UseMvcWithDefaultRoute();

This code handles a very specific URL pattern inline as part of the configuration code. I could have also handled this with a custom MVC route, but it’s arguably easier and cleaner to set up a custom IRouterService route to handle a very specific, non-application concern like this as part of the application configuration process.

### Http.Sys Support

For Windows, ASP.NET Core 2.0 now also supports Http.sys as another Web server in addition to the Kestrel and IIS/IIS Express servers that are supported by default. **Http.sys** is the kernel driver used to handle HTTP services on Windows. It's the same driver that IIS uses for all of its HTTP interaction, and now you can host your ASP.NET Core applications directly on Http.sys using the **Microsoft.AspNetCore.Server.HttpSys** package.

The advantage of using Http.sys directly is that it uses the Windows kernel HTTP infrastructure, which is a hardened Web Server front-end that supports high-level support for SSL, content caching and many security-related features not currently available with Kestrel.

For Windows, the recommendation has been to [use IIS as a front-end reverse proxy](https://weblog.west-wind.com/posts/2016/Jun/06/Publishing-and-Running-ASPNET-Core-Applications-with-IIS) in order to provide features like static file compression and caching, SSL management, and rudimentary connection protections from various HTTP attacks against the server.

By using the Http.sys server, you can get most of these features without having to use a reverse proxy in front of Kestrel, which has a bit of overhead.

To use Http.sys, you need to explicitly declare it using the **.UseHttpSys()** configuration added to the standard startup sequence (in **program.cs**):

WebHost.CreateDefaultBuilder(args)

.UseStartup<Startup>()

.UseHttpSys(options =>

{

options.Authentication.Schemes =

AuthenticationSchemes.None;

options.Authentication

.AllowAnonymous = true;

options.MaxConnections = 100;

options.MaxRequestBodySize = 30000000;

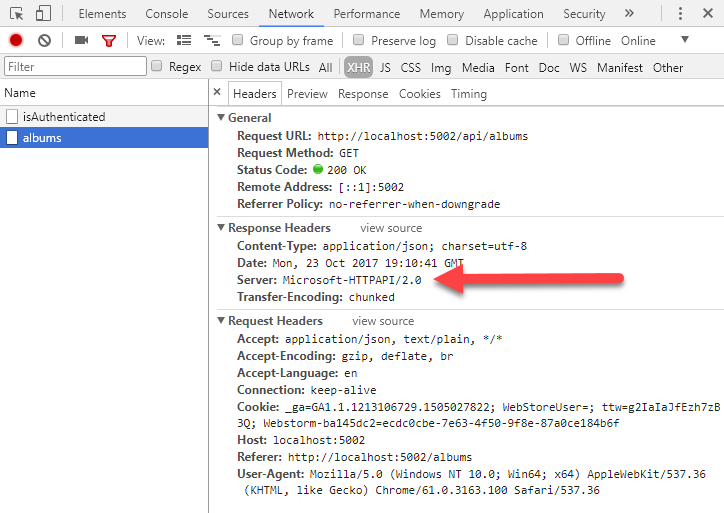
options.UrlPrefixes

.Add("http://localhost:5002");

})

.Build();

You can then configure the local port to make it accessible both locally and remotely (by opening up a port on the firewall). When you do, you’ll see the HTTP Sys server as part of the response headers, as shown in **Figure 6**.



**Figure 6:** Http.sys hosting in ASP.NET Core provides efficient Windows hosting without a proxy

Microsoft has a detailed document that describes how to set up http.sys hosting at <https://goo.gl/XnFz88>.

If you’re running on Windows, this might be a cleaner and more easily configurable way to run ASP.NET Core applications than doing the Kestrel > IIS dance. Doing some quick over-the-finger performance tests with **West Wind WebSurge** ([websurge.west-wind.com](https://websurge.west-wind.com/)) show that running with raw Http.sys is a bit faster than running Kestrel with IIS as a front-end proxy because it eliminates the Proxy hop. However, by cutting out IIS, your application is now responsible for serving all content rather than letting IIS handle standard static content.

For a public facing website, you're probably better off with full IIS, but for raw APIs or internal applications, Http.sys is a great alternative for Windows-hosted server applications.

### RazorPages

In ASP.NET Core 2.0, Microsoft rolls out **RazorPages**. RazorPages is completely new, although it's based on concepts that should be familiar to anybody who's used either ASP.NET WebPages or—gasp—even WebForms that came before it.

RazorPages provide a new approach for creating scripted server-side HTML content, using an approach similar to client-side frameworks like Angular or Vue.

When I first heard about RazorPages a while back, I had mixed feelings about the concept. Although I think that a script-based framework is an absolute requirement for many websites that deal primarily with content, I also feel that requiring a full ASP.NET Core application setup with controllers and views separated out, plus a full site deployment process just to run script pages is a bit of overkill. One of the advantages of tools like WebPages and WebForms is that you don't have to “install” an application and you just drop a new page into a server and run. With RazorPages, you’re still installing and deploying a Web “application.”

But think about how much clutter is involved in MVC to get a single view fired up in the default configuration that ASP.NET MVC projects use:

* Controller Class
* Controller Method that invokes your View (Controllers folder)
* View Model (Models Folder)
* View (View/Controller folder)

In other words, code in MVC is scattered all over the place and there’s quite a bit of ceremony to get a page rendered. MVC works great for real “applications” that require extensive logic to display pages. But MVC is terrible for mostly static content pages that only have minimal server logic embedded in pages like landing pages, company or product and news sites, etc. that require a little bit of dynamic logic. This type of content is much better served by self-contained script pages.

RazorPages mostly addresses this scenario, but at the same time, it’s also powerful enough to create a much more complex application with an optional fresh new PageModel class that allows attaching multiple **actions** based on HTTP verbs. This optional `CodeBehind` like approach has common startup code and specifics for GET and POST operations, for example. This is a different way of doing MVC that has much more in common with client-side frameworks like Angular or Vue than ASP.NET MVC.

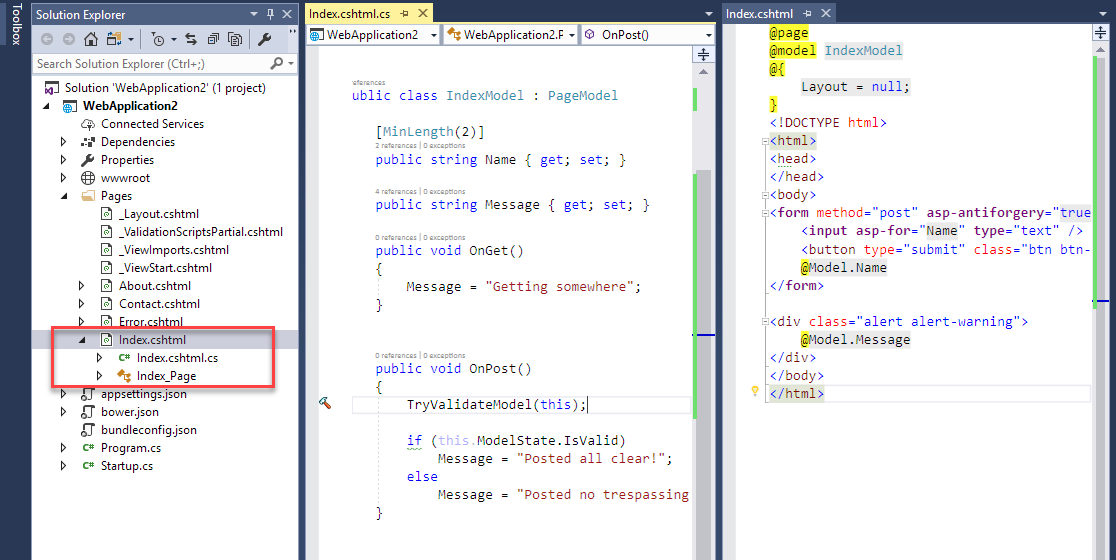
One thing to understand is that RazorPages still requires an ASP.NET Core application to be deployed on the server. Unlike WebForms or WebPages, which ”just work” by dropping a page into a virtual folder or website, RazorPages are hosted inside of their own ASP.NET Core application, which means you still have to build and deploy that application. Once the application is up however you can simply add or update Razor Pages while the application is running as Razor Pages are dynamically compiled at runtime. Because Razor Pages is part of an application, it also provides the benefits of being able to create custom components and take part of the ASP.NET Core Middleware pipeline and everything else that ASP.NET Core’s base functionality provides. It’s up to you to decide what you want to support.

RazorPages use the same exact Razor syntax that MVC uses, so unlike the older WebPages technology, RazorPages and MVC Views share the same Razor engine and parsers and you can move Views between the two models easily. If you really want to get fancy, you can even mix and match and use RazorPages and MVC in the same application. A common scenario might be to use RazorPages for HTML content and MVC for API requests.

**Listing 5** shows an example of a self-contained scripted Razor page that includes a model and controller methods inline.

Before you completely dismiss inline code in the **.cshtml** template, consider that code inside the RazorPage is dynamically compiled at runtime, which means that you can make changes to the page without having to recompile and restart your entire application as you have to do with controller or even CodeBehind code in ASP.NET Core applications.

Although I really like the fact that you can embed a model right into the Razor page for simple pages, this gets messy quickly. More commonly, you pull out the PageModel into a separate ‘CodeBehind’ PageModel class. The default template that creates a RazorPage in Visual Studio does just that. When you create a new RazorPage in Visual Studio, you get a **.cshtml** and a nested **.cshtml.cs** file, as shown in **Figure 7**.



**Figure 7:** RazorPage ‘CodeBehind` uses a hybrid model/controller PageModelclass

The **PageModel** subclass in this scenario becomes a hybrid of controller and model code very similar to the way many client-side frameworks like Angular handle the MV\* operation, which is more compact and easier to manage than having an explicit controller located in yet another external class.

**PageModel** supports implementation of a few well-known methods, like **OnGet()**, **OnPost(),** etc., for each of the supported verbs that can handle HTTP operations just like you would in a controller. An odd feature called **PageHandlers** using the **asp-page-handler="First"** attribute lets you even further customize the methods that are fired with a method postfix like **OnPostFirst()**, so that you can handle multiple forms on a single page for example.

Although traditional MVC feels comfortable and familiar, I think RazorPages offers a viable alternative with page-based separation of concerns in many scenarios. Keeping View and View-specific Controller code closely associated usually makes for an easier development workflow and I'd already moved in this direction with feature folder setup in full MVC anyway. If you squint a little, the main change is that there are no more explicit multi-concern controllers, just smaller context specific classes.

RazorPages is not going to be for everyone, but if you're like me and initially skeptical, I encourage you to check them out. It's a worthwhile development flow to explore and for the few things I still use server-generated HTML for, I think RazorPages will be my tool of choice on ASP.NET Core HTML content.

### Tooling Issues Have Been Mostly Resolved

I've talked about a lot of things that are improved and that make 2.0 a good place to jump into the fray for .NET Core and ASP.NET Core. But it's not without its perils: there are still a few loose ends, especially when it comes to tooling.

When I wrote about problem issues three months ago in a blog post (<https://goo.gl/qX8R6Z>), there were major tooling issues with SDK projects that made working with .NET Core and .NET Standard projects pretty painful. Slow compiles, terribly slow run/debug/restart cycles, incorrect and non-clear error information in Visual Studio and other tools, slow **dotnet run watch** recycles made for a pretty dreadful developer experience. I’m happy to say that most of the performance and stability issues I griped about in that original blog post have been addressed in subsequent SDK and Visual Studio updates.

For this reason, I highly recommend that you make sure you’re running the latest version of the .NET SDK (2.12 at the time of writing) and Visual Studio 2017 (vs2017.5.5 at time of writing) or OmniSharp. If you’ve experienced these slow and unstable tooling issues and stepped away from .NET Core/Standard because you thought it was just too unstable, it might be time to give it another try. Faster and reliable tooling makes a world of difference in how you experience the platform.

In the three months since my last blog post, performance of the tooling has improved drastically and overall stability of the compiler tools in Visual Studio, as well as other tools like Visual Studio Code using OmniSharp as well as JetBrains’ new Rider IDE, is vastly better. Also gone are the constant invalid compiler errors and false-positive error messages. Tests now run reasonably fast and both Test Explorer and Resharper’s Test Runner can now clearly display tests for multiple build targets.

The tooling is clearly on the right path to being on par with full framework tools in terms of stability and performance, but we’re not quite there yet. There are still rough edges, but for now, I’d at least declare it as **good enough to go**.

### Go Ahead and Jump

The entire 2.0 train of upgrades is a **huge** improvement over what came in V1.x, and the progress of change has been pretty rapid and amazing. I—and many others I know—were sitting on the fence with .NET Core/Standard for a long time. It was easy to see the promise of the .NET Core/Standard long ago, but execution and implementation in V1 left too many holes that made jumping in seem too daunting or simply not enough of an incentive. For v1.x, I was sitting on the fence, dabbling, watching, learning, and I have to say that I’m glad I waited out the initial releases before doing any “real” work.

But with v2.0, I’ve jumped in with both feet, converting old full-framework libraries to .NET Standard and ASP.NET Core versions, as well as starting a number of internal ASP.NET Core 2.0 projects. My first work has been around libraries, which has been a great learning experience, and the experience has been a good one. In fact, the benefits of the new project system and multi-targeting has been a big improvement over older versions. I’ve also finally started a few ASP.NET Core Web projects for some internal projects that are in dire need of updates, and using many of the new features in ASP.NET Core has been a pure joy. There are still challenges because there are many changes in how things work and rediscovering where some features live (who moved my cheese syndrome), but overall, the reimagining of the ASP.NET platform is a vast improvement.

Another thing that I find extremely promising is that Scott Hunter recently mentioned that .NET Core 2.0 and .NET Standard 2.0 will stay put for a while, without major breaking changes moving forward. I think this is a great call: I think we all need a breather instead of chasing the next pie in the sky. We can all use some time to try to catch up to the platform, figure out best practices, and also see what things still need improving.

It's been a rough ride getting to 2.0 and I, for one, can appreciate a little stretch of smooth road ahead.

Listing 1: The new, simpler SDK Style Project Format

<Project Sdk="Microsoft.NET.Sdk.Web">

<PropertyGroup>

<TargetFramework>netcoreapp2.0</TargetFramework>

<UserSecretsId>d900d6cb-0e21-403b-94a3-17412045e7b4  
 </UserSecretsId>

<Version>2.0.0</Version>

</PropertyGroup>

<ItemGroup>

<PackageReference Include="Microsoft.AspNetCore.All"

Version="2.0.0" />

<ProjectReference Include="..\AlbumViewerBusiness\AlbumViewerBusiness.csproj" />

</ItemGroup>

<ItemGroup>

<Content Update="wwwroot\\*\*\\*;Areas\\*\*\Views;\_

appsettings.json;album.js;web.config">

<CopyToPublishDirectory>PreserveNewest</CopyToPublishDirectory>

</Content>

</ItemGroup>

<ItemGroup>

<Content Include="albumsdata.json">

<CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>

</Content>

</ItemGroup>

<ItemGroup>

<DotNetCliToolReference Include="Microsoft.DotNet.Watcher.Tools"

Version="2.0.0" />

</ItemGroup>

</Project>

Listing 2: A multi-targeted SDK Style Project

<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>

<TargetFrameworks>netstandard2.0;net45</TargetFrameworks>

</PropertyGroup>

<PropertyGroup Condition="'$(Configuration)'=='Debug'">

<DefineConstants>TRACE;DEBUG;</DefineConstants>

</PropertyGroup>

<PropertyGroup Condition=" '$(Configuration)' == 'Release' ">

<DefineConstants>RELEASE</DefineConstants>

</PropertyGroup>

<ItemGroup>

<PackageReference Include="Newtonsoft.Json" Version="10.0.3" />

</ItemGroup>

<ItemGroup Condition=" '$(TargetFramework)' == 'netstandard2.0'">

<PackageReference Include="System.Data.SqlClient"

Version="4.4.0" />

</ItemGroup>

<PropertyGroup  
 Condition=" '$(TargetFramework)' == 'netstandard2.0'">

<DefineConstants>NETCORE;NETSTANDARD;NETSTANDARD2\_0

</DefineConstants>

</PropertyGroup>

<ItemGroup Condition=" '$(TargetFramework)' == 'net45' ">

<Reference Include="mscorlib" />

<Reference Include="System" />

<Reference Include="System.Core" />

<Reference Include="Microsoft.CSharp" />

<Reference Include="System.Data" />

<Reference Include="System.Web" />

<Reference Include="System.Drawing" />

<Reference Include="System.Security" />

<Reference Include="System.Xml" />

<Reference Include="System.Configuration" />

</ItemGroup>

<PropertyGroup Condition=" '$(TargetFramework)' == 'net45'">

<DefineConstants>NET45;NETFULL</DefineConstants>

</PropertyGroup>

</Project>

Listing 3: Configuration is a lot more organized in ASP.NET Core 2.0

public void ConfigureServices(IServiceCollection services)

{

services.AddDbContext<AlbumViewerContext>(builder =>

{

var connStr = Configuration["Data:SqlServerConnectionString"];

builder.UseSqlServer(connStr);

});

services

.AddAuthentication(

CookieAuthenticationDefaults.AuthenticationScheme)

.AddCookie(o =>

{

o.LoginPath = "/api/login";

o.LogoutPath = "/api/logout";

});

services.AddCors(options =>

{

options.AddPolicy("CorsPolicy",

builder => builder

.AllowAnyOrigin()

.AllowAnyMethod()

.AllowAnyHeader()

.AllowCredentials());

});

// Add Support for strongly typed Configuration and map to class

services.AddOptions();

var sec = Configuration.GetSection("Application");

services.Configure<ApplicationConfiguration>(sec);

}

public void Configure(IApplicationBuilder app)

{

app.UseAuthentication();

app.UseCors("CorsPolicy");

app.UseDefaultFiles();

app.UseStaticFiles();

app.UseMvcWithDefaultRoute();

}

Listing 4: A self-contained ASP.NET Core App using IRouterService

public static class Program

{

public static void Main(string[] args)

{

WebHost.CreateDefaultBuilder(args)

.ConfigureServices(s => s.AddRouting())

.Configure(app => app.UseRouter(r =>

{

r.MapGet("helloWorldRouter/{name}",

async (request, response, routeData) =>

{

var result = new

{

name = routeData.Values["name"] as string,

time = DateTime.UtcNow

};

await response.Json(result);

});

r.MapPost("helloWorldPost"

async (request, response, routeData) => {

...

};

}))

.Build()

.Run();

}

public static Task Json(this HttpResponse response, object obj,

Formatting formatJson = Formatting.None)

{

response.ContentType = "application/json";

JsonSerializer serializer = new JsonSerializer

{

ContractResolver = new CamelCasePropertyNamesContractResolver()

};

serializer.Formatting = formatJson;

using (var sw = new StreamWriter(response.Body))

using (JsonWriter writer = new JsonTextWriter(sw))

{

serializer.Serialize(writer, obj);

}

return Task.CompletedTask;

}

}

Listing 5: RazorPages lets you embed

@model IndexModel

@using System.ComponentModel.DataAnnotations

@using Microsoft.AspNetCore.Mvc.RazorPages

@functions {

public class IndexModel : PageModel

{

[MinLength(2)]

public string Name { get; set; }

public string Message { get; set; }

public void OnGet()

{

Message = "Getting somewhere";

}

public void OnPost()

{

TryValidateModel(this);

if (this.ModelState.IsValid)

Message = "Posted all clear!";

else

Message = "Posted no trespassing!";

}

}

}

@{

Layout = null;

}

<!DOCTYPE html>

<html>

<head>

</head>

<body>

<form method="post" asp-antiforgery="true">

<input asp-for="Name" type="text" />

<button type="submit">Show Hello</button>

@Model.Name

</form>

<div class="alert alert-warning">

@Model.Message

</div>

</body>

</html>

.NET Core and .NET Standard

.NET Standard 2.0 is a **specification**, or a blueprint of features that have to be implemented by a specific platform. .NET Core 2.0 is a specific platform that **implements** the .NET Standard 2.0, which means that it has all the APIs required by Standard. Because .NET Core 2.0 implements .NET Standard 2.0, it can consume any libraries that are targeted to .NET Standard 2.0 or earlier. They’re backward compatible.

The .NET Standard includes a fixed number of APIs, which any platform that wants to implement the specification **has to** implement. The idea is that you have many concrete platforms that **support** .NET Standard and any library that implements that version of the Standard—or an earlier one—can then **run on any of these platforms**. This is great for library developers who can now build libraries that run on most of the .NET Platforms, including mobile platforms like Xamarin or other platforms like Xamarin.Mac.

Implementing .NET Standard 2.0

There are **many** .NET Standard 2.0 implementations: .NET Core 2.0, .NET 4.61, .NET 4.71, Xamarin for iOS 10.14 all **implement** .NET Standard 2.0. Note that these are very **specific** versions. Each of those platforms can consume a library that was built for .NET Standard 2.0 (or earlier) as well as platform-specific libraries for the specific platform. A .NET 4.61 application can consume both .NET Standard packages as well as .NET 4.5 assemblies, for example, and .NET Core 2.0 applications can use .NET Standard 2.0 and .NET Core assemblies, although I suspect that will be rather rare.

When you build class libraries going forward, you’ll want to target .NET Standard to make the library available on as many platforms as possible. Top-level application projects such as an ASP.NET Core project, .NET Core or full Framework Console apps will target a specific runtime. However, because that runtime likely implements .NET Standard 2.0, it can then access any of the .NET Standard libraries.

.NET Core Runtimes

With .NET Core 2.0, Microsoft decided to go back to distributing system-level runtimes that are pre-installed much in the way that the full .NET Framework was installed in the past.

The big difference with the .NET Core runtimes is that each runtime is versioned and installed side-by-side with other versions. Each version can exist independently of any other version, so there are no version conflicts.

This differs from full framework runtimes, which were rev’d very infrequently and are updated in place, resulting in potential version issues.

Applications can specify exactly which version of the .NET Core runtime they want to target when the application is built and assuming that the runtime is installed on the local or deployed computer, the application can then run using that runtime.

Get .NET Core Help for Free

Looking to create a new or convert an existing application to .NET Core 2.0 or ASP.NET Core 2.0? Get started with a FREE hour-long CODE Consulting session to make sure you get started on the right foot. Our consultants have been working with and contributing to the .NET Core and ASP.NET Core teams since the early pre-release builds. Leverage our team’s experience and proven track record to make sure your next project is a success. For more information visit [www.codemag.com/consulting](http://www.codemag.com/consulting) or email us at [info@codemag.com](mailto:info@codemag.com" \t "_blank)