



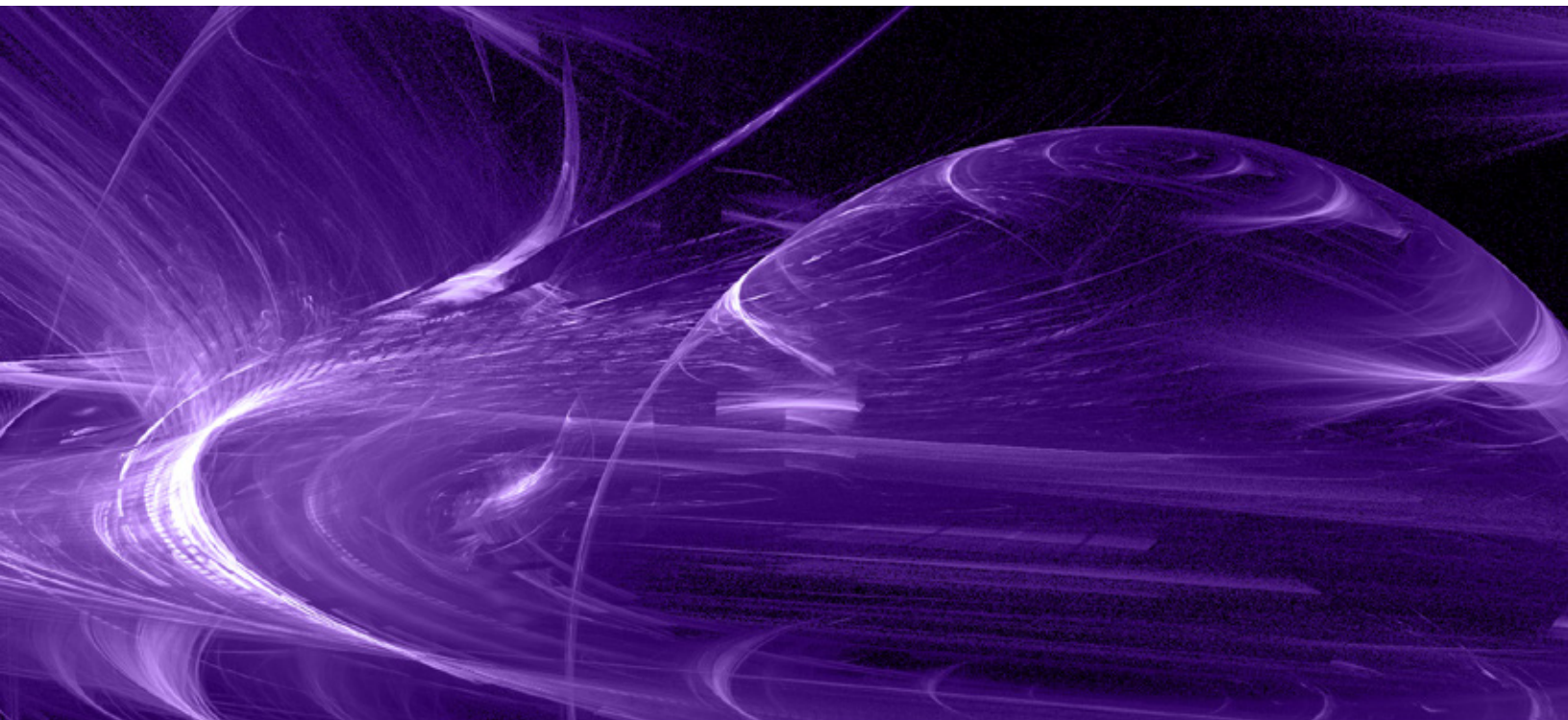
STACKIFY RETROSPECTIVE

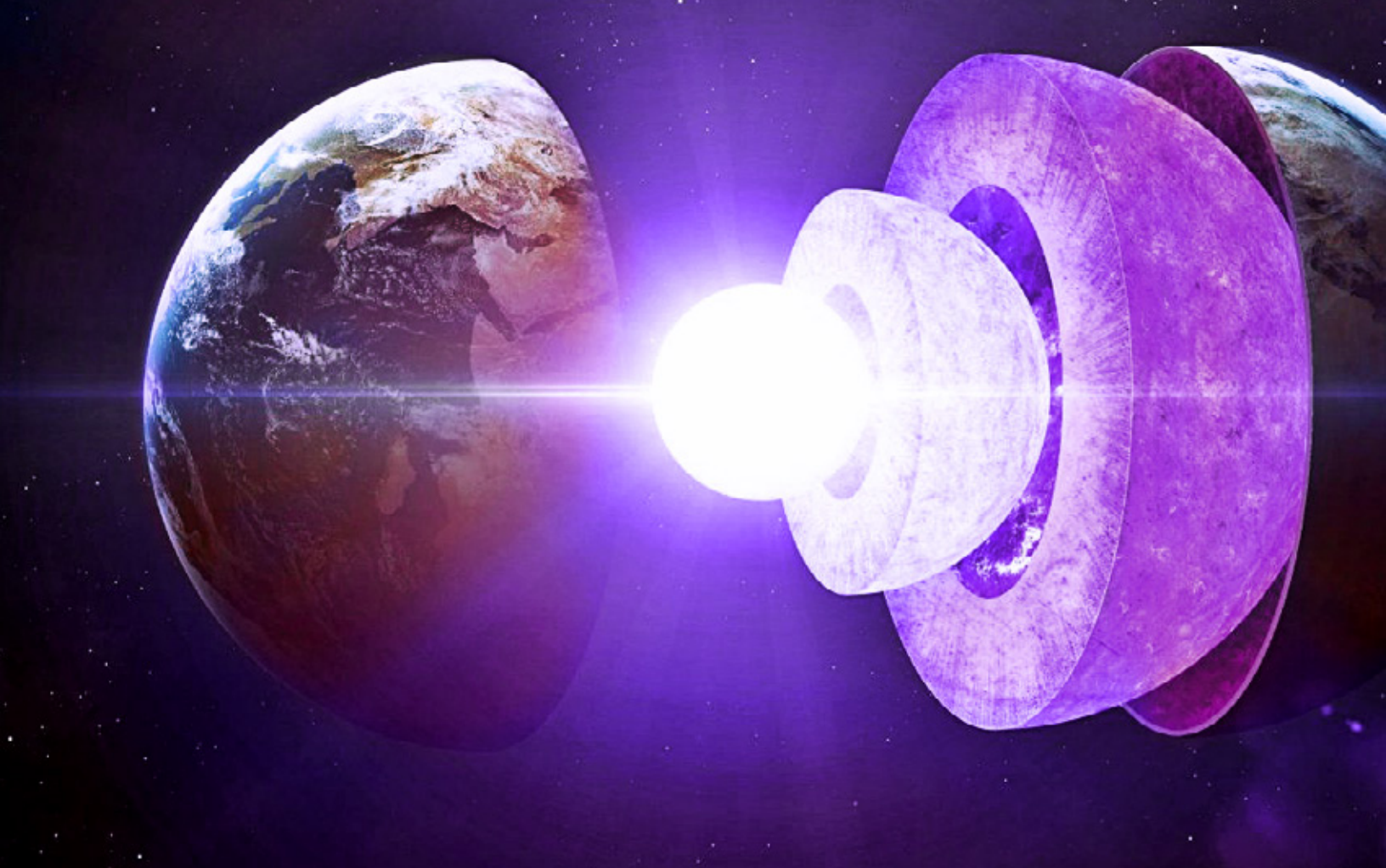
# Essential .NET Core:

Methods for Optimizing Web Apps in Azure,  
Logging, Deploying to IIS, and More

# Table of Contents

<u>Use Cake to Build .NET Core Apps &amp; Version Assemblies .....</u>	5
<u>Create .NET Core Windows Services With Visual Studio '17.....</u>	12
<u>Master Logging for .NET Core .....</u>	16
<u>Deploy to and Optimize .NET Core Web Apps in Azure.....</u>	27
<u>Deploy ASP.NET Core to IIS with .NET Core Hosting.....</u>	36
<u>A Retrospective on Converting to .NET Core .....</u>	41





# Introduction

At Stackify, we build with developers in mind. We experience the same frustrations, hurdles and successes as our customers because we live and breathe code. In our Retrospectives series, we pool together all our knowledge and experience on the most relevant and integral topics in the community and deliver them to developers. We've done the research so you can do the coding.

In our Essential .NET Core Methods guide, we share the lessons that we've learned, from converting to .NET Core to deploying and optimizing our apps. Stackify has been working with .NET Core for over a year now, and we have been very impressed with it. It is amazing to see what Microsoft and the community have done with it. We hope this guide will further emphasize the versatility and efficiency of .NET Core, while helping developers in their day-to-day work.

*"Until Stackify, there was no one tool to do monitoring, performance dashboards, log aggregation, APM and more. This same technology would come with half a dozen or more products and even more expense. Stackify provides all of this functionality*



*out of the box, in one package, and provides you the actionable insights necessary to target problems when and where they happen."*

*—Jake Dubin, VeriShip*



APM



LOGS



ERRORS



MONITORING



QA/PRE-PROD



PRODUCTION

STARTING AT ONLY

**\$10**/mo

# Use Cake to Build .NET Core Apps

## Let Them Eat Cake!

We like to begin with the end in mind at Stackify. As our platform started to unfold into its first compiled bits nearly five years ago, it quickly became apparent that we needed some decent structure around how we built and deployed the many layers of our complex stack.

Like most shops, we were in search of an ALM system that could encompass our sprint/work item management and build services. We wanted something SaaS-based as well, and ultimately we landed with the Atlassian JIRA suite. Their CI server, Bamboo, has orchestrated our builds and deploys ever since then.

Doing a build on your build server and workstation should be exactly the same. There have been far too many times where building locally doesn't have the same result as building on the CI server. And getting the CI server to build and behave the way we want it to is always less than ideal.

This all came to a head a few months ago when we started converting [Prefix](#), our ASP.NET profiler, to .NET Core. We needed our build server to:

- Pull the Git repo
- Version the assembly
- Check in the modified assembly version change
- Build it – now with the .NET CLI, and not MSBUILD
- Nupack it
- Ship to our Myget server
- Pull request this release branch back to 'master' in our repo

Traditionally, we'd do this with a series of Powershell Script Tasks in Bamboo, but there's one big pain point: we use AWS Elastic EC2 instances for build agents, and to test even the slightest change you need to create a new image and spin it up in Bamboo and have it build.

## Enter Cake!

We needed a task runner that would allow anyone on our dev team to execute all of these steps locally in just the same way our build server would, both to validate the process works correctly, and in the event our build server was down or had other issues.

We came across CakeBuild and read the docs. It just made sense. With Cake, you get

- A C# based DSL (no more Powershell or command line scripting!!)
- A huge library of built-in scripts for nearly any task you would need
- A system that supports third party scripts
- It's all nuget-based; it pulls in everything it needs
- Good support for .NET Core

On paper, it was perfect. So, let's take a look at what it does. Here's how we accomplished some of the specifics, with a big focus on assembly versioning.

## Task: Version Assemblies

But first, a word on our versioning methodology. With .Net Core, you can keep the assembly version in project.json, which makes a lot of sense because it's easy to manipulate. It's just a JSON string. When we are prepping a new release, we first cut a "release candidate" branch in Git, and it will be named with the target version, i.e:

**root/rc/1.2**

We will do our testing and, once done, cut the actual release branch, i.e.

**root/release/1.2**

If we have to release a fix for the version, we work on that same branch. The next builds will be 1.2.2, 1.2.3, 1.2.4, etc.

For all other builds, the version number will follow the default format in .NET Core, i.e: 1.2.0-\*, where "\*" gets replaced with the branch name + build number, so you'd end up with a 1.2.0-develop-1, or 1.2.2-rc-3, etc. This allows us to push a nuget package and dll that that is testable but never ever will be confused with a release version. If you use any pre-release .NET Core packages, this will seem familiar.

So, here is what a Cake task looks like to version:

```
var target = Argument("target", "Default");
```

```
var configuration = Argument("configuration", "Release"); var buildVersion = ""; var buildSuffix = ""; var branch = ""; bool bambooRelease = false; Task("Version").Does(() => { branch = (EnvironmentVariable("bamboo_planRepository_branchName") ?? "unknown"); var buildNumber = (EnvironmentVariable("bamboo_buildNumber") ?? "0"); var json = ParseJsonFromFile("project.json") var version = json["version"].ToString();
```

See that? By checking the environment variables (via a built in script), we get the branch name if we're running in Bamboo, otherwise we don't care, and "Unknown" is substituted (we could still get it with a Git task, but it wasn't necessary here). Also parse the project.json again with a built in script.

Next, use another built in script to check to see if we are, in fact, running in Bamboo.

```
if(Bamboo.IsRunningOnBamboo){
    Information("Running on Bamboo");
    Information(version);
    Information(branch);
    Information(buildNumber);

    if(branch.Contains("release/"))
    {
        bambooRelease = true;
        Information("Plan is release");
        var relVer = branch.Split('/')[1];
        var newVer = relVer + "." + buildNumber + "-*";
```

Here we look for the branch name to support our versioning. If it is release, we know the next part is our Major.Minor. We append on the build number, and serialize the JSON back out to the project.json file.

```
//only write back if it is a release build
    json["version"]=newVer;
    SerializeJsonToFile("project.json",json);

}
Else{
```



For non-release, it will generate something like "1.2.0-rc-1" and in this specific case, we don't need to write it back out to the file.

```
Information("Plan is not release");
    var cleanBranchName = branch.Replace("/", "-");
    buildSuffix = cleanBranchName + "-" + buildNumber;
    var newVer = version.Replace("*, cleanBranchName + "-" +
buildNumber);
    buildVersion = newVer;
}

}
else{
    Information("Local Build");
    var cleanBranchName = "local";
    buildSuffix = cleanBranchName + "-" + buildNumber;
    var newVer = version.Replace("*, cleanBranchName + "-" +
buildNumber);
    buildVersion = newVer;
}
Information(buildVersion);
Information(buildSuffix);

});
```

And that's it: assembly versioning done easily and flexibly with CakeBuild.

## Task: Nuget Package

The next step to build and pack is just as easy:

```
Task("Pack").Does(() => {
    if(buildSuffix!=""){
        DotNetCorePack("project.json", new DotNetCorePackSettings{ Configuration
= configuration, VersionSuffix = buildSuffix });
    }
    else{
        DotNetCorePack("project.json", new DotNetCorePackSettings{ Configuration =
configuration });
    }

    var outputdir = "bin\\"+configuration;
    CopyFileToDirectory("NugetPack.ps1",outputdir);
    var wd = DirectoryPath.FromString(outputdir);

    if(Bamboo.IsRunningOnBamboo)
    {
        StartProcess("powershell.exe", new ProcessSettings { Arguments = "--file
NugetPack.ps1", WorkingDirectory = wd});
        StartProcess("powershell.exe", new ProcessSettings {Arguments = "--file
MygetPush.ps1", WorkingDirectory = wd});
    }
});
```

The DotNetCorePack script actually builds the project and the nuget package in one step. It runs a build using the .NET CLI, which means that if we're targeting multiple frameworks, they will all be in the package, so we can reference in projects that target NETStandard, Netcoreapp, and full .NET framework. All in one simple step.

The NugetPack.ps1 file is a script that we actually created and have checked into the project. It simply checks for the existence of a Nuget package (and symbols package) and then generates MygetPush.ps1 which is a script that will actually push the package to our private Myget server. It only does this for Bamboo builds, which means that if we run this Cake script locally, we're not going to end up with a bunch of superfluous packages on our build server. You could put all of this logic directly in the Cake script if you wanted to; we already had those scripts from our previous process, so we just left it.

## Task: Commit and Push

The last step is to commit and push the change to our project.json file.

```
Task("CheckinVersion").Does(() => {

    if(bambooRelease){
        var repPath = DirectoryPath.FromString(".");
        var filePaths = new FilePath[] { ".\\project.json" };
        GitAdd(repPath, filePaths);
        GitCommit(repPath, "Bamboo Build Task", "bamboo@stackify.com", "Bamboo has detected a release build - versioning assembly");

        GitPush(repPath);

        var repositoryDirectoryPath = DirectoryPath.FromString(".");
        var gitBranch = GitBranchCurrent(repositoryDirectoryPath);
        var remote = (gitBranch.Remotes[0].PushUrl).Split('/');
        var repName = remote[remote.Count()-1];

        string json = "{\\\"destination\\\": { \\\"branch\\\": { \\\"name\\\": \\\"master\\\" } }, \\\"source\\\": { \\\"branch\\\": { \\\"name\\\": \\\""+branch+"\\\" } }, \\\"title\\\": \\\"Merge Bamboo Build\\\" }";

        StartProcess("C:\\curl\\bin\\curl.exe", new ProcessSettings { Arguments = "-v -u [our user name] \"[git repo url]"+repName+"/pullrequests\" -H \"Content-Type: application/json\" -d \" "+ json + "\""});
    }
});
```

Note that we redacted some details, but again, used the built-in scripts that Cake provides to commit and check in any file changes on our current branch (which happens to be my modified project.json with the new version number). Then use CURL to post a Pull Request to the API of my Bitbucket repo. The only part of this that is painful was the escape characters needed for the JSON that gets posted.

## Putting the Icing on the Cake

One thing that you've probably noticed in this is that we're still using Bamboo. That's because Cake is simply a DSL for executing tasks. We still need an orchestration engine (like Bamboo) to kick off and execute the Cake builds. But Bamboo is now rather agnostic about the build. We've gone from having 10 or so complicated, hard to test Bamboo tasks to two simple tasks:

1. Pull repo.
2. Execute build.ps1

We've also achieved a lot of portability and could very easily migrate to a different CI server with a small level of effort.

If you think this can help make your life easier, check out CakeBuild. Don't stop with just the build, either. Cake has scripts and add-ins for deployment (IIS, Azure, Docker, etc), for SQL tasks, to work with TFS work items, NUnit, etc. And if it doesn't support something you need to do, you can simply create it yourself and publish as an add-on.

3.0

Now Available!

### Prefix

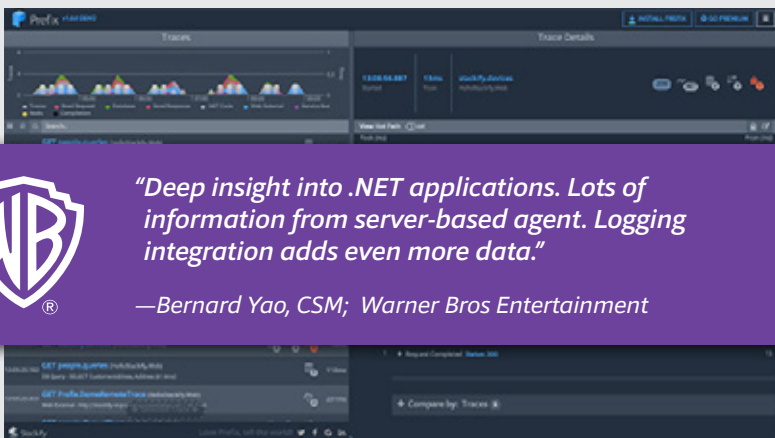
*Prefix is a lightweight tool that shows .NET developers:*

**Logs + Errors + Queries + More**  
*in real time—and it's free!*



*"Deep insight into .NET applications. Lots of information from server-based agent. Logging integration adds even more data."*

—Bernard Yao, CSM; Warner Bros Entertainment



# Create .NET Core Windows Services With Visual Studio 2017

Mobile and web applications might be all the rage these days, but a lot of the real work still gets done via background services and scheduled tasks. At Stackify, we use a lot of background .NET services for processing all of the data that our systems ingest. One of the common problems with .NET Core when it first came out was how to create Windows Services. Let's focus on how to take code that was designed to be portable with .NET Core and use it in a Windows Service.

## The Problem:

### .NET Core Does Not Like Windows Specific "Stuff"

The strength of .NET Core is the portability of your code across multiple types of apps and operating systems. This requires a unified surface area of APIs. Anything too OS specific is not supported. For example, Windows registry, WMI, Microsoft specific cryptography APIs, and... Windows Services are some quick examples. Across the entire .NET Core framework there is tiny trade-offs and differences here and there based on Windows vs Linux.

**Windows Services only work with the full .NET Framework. The good news is you can mix .NET projects that target .NET Core and the full .NET Framework.**

You effectively have 2 options now when creating a Windows Service:

1. Standard .NET 4.5 Windows App referencing .NET 4.5 packages/assemblies
2. Standard .NET 4.5 Windows App referencing .NETStandard packages

For the last few years, all .NET code has been written with .NET 4.5 and basic assembly references or NuGet packages. With .NET Core, we now reference packages based on the .NETStandard. This is what enables the cross-platform capabilities.

Most popular 3rd party libraries have all switched to the new .NETStandard. Common libraries like JSON.NET, log4net, NLog, Dapper, Entity Framework and many others now support it. Our own StackifyLib package for sending application errors, logs, and metrics to our platform has been converted. You are likely these days to be referencing .NETStandard packages and you don't even know it!

**The key to making a .NET Core Windows Service** is ensuring your code is in a shared class library that targets .NETStandard. You can then use that code in non-Windows apps, ASP.NET, and Windows Services. If you also need to write some Windows specific code, you can do that directly in the Windows Service.

Our product, Prefix, is a good example of this. We converted it to .NET Core so that we could support .NET and Java on Windows, but then also make a version for Java on the Mac. All of the core code for the app is in shared class libraries. Our Windows Service uses those libraries and we have some Windows specific code in a separate library. For us, that includes some special code for WMI, IIS, etc.

## Creating .NET Core Windows Services

### Warning About Visual Studio 2015

Visual Studio 2015 does not allow project references between .NET Core (xproj) and .NET Framework (csproj) projects. If you are working with .NET Core, we highly recommend upgrading to Visual Studio 2017 so you can use all the latest tooling. There are a lot of changes around the project files.

### Mixing .NET Framework & .NET Core with Visual Studio 2017

With Visual Studio 2017, all projects now use csproj files again. You can easily create project references between your projects. Windows Services are much easier to create with 2017.

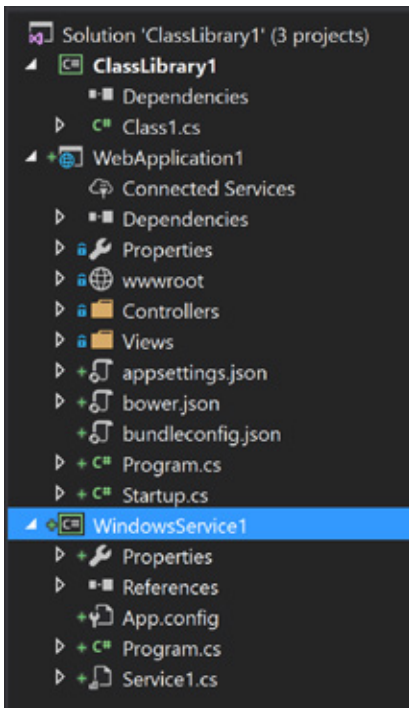
If you plan on deploying to Windows and Linux, we recommend putting your code in a separate project that acts as a shared class library. You can then access this code from an ASP.NET application, console application, or Windows Service. This would give you the flexibility to deploy your code and applications to Windows or Linux with maximum flexibility.

Treat your Windows Service as basically a shell, and putting all of your key application logic in the shared .NETStandard library. If you have code that is very Windows specific, you can put that code directly in the Windows Services project.

## Example Project

We've created a really simple sample project to walk through how to make all this work nicely. This includes both a .NET Core ASP.NET app and Windows Service. They both reference a shared class library. This shows off how to mix .NET Core and .NET Framework projects together.

**Note:** You are able to make a Windows Service exactly the same way you always have as long as your .NET Core code is in a .NETStandard library. We will skip going into any real details about what a Windows Service is or how to make one in detail. This article is focused on getting .NET Core code to work with your Windows Service.



After creating all three projects here are the default frameworks that Visual Studio 2017 is using for all three:

- WindowsService1 – .NET Framework 4.5.2
- WebApplication1 – .NETCoreApp1.1
- ClassLibrary1 – .NETStandard 1.4

We modified both by our web app and Windows Service to reference ClassLibrary1. Adding the project reference worked just fine, but we cannot build, and get this error below:

```
'C:\BitBucket\CoreWindowsServicesExample\ClassLibrary1\ClassLibrary1.csproj' targets '.NETStandard,Version=v1.4'. It cannot be referenced by a project that targets '.NETFramework,Version=v4.5.2'.
```

This means that our Windows Service can't reference the class library based on the current target frameworks. A .NET Framework 4.5.2 app can't reference a class library that is targeting .NETStandard 1.4. They are not compatible. There is a simple fix, though.

To fix the compilation error, we changed our Windows Service project to target .NET Framework 4.6.1 and now it works fine!

This article has the matrix for compatibility: <https://docs.microsoft.com/en-us/dotnet/articles/standard/library>.

The ASP.NET web app that targets .NETCoreApp1.1 works perfectly with .NETStandard with no changes needed. Now we have a shared class library that can be used in an ASP.NET web app on Linux plus a Windows Service!

Currently, the class library is our example using .NETStandard 1.4. You likely will want to target 1.5 to have maximum compatibility with other libraries that you may want to use. If you change your Windows Services project to target .NET Framework 4.6.1, it will then work class libraries targeting .NETStandard 1.5.

## DLL Hell is Gone. Welcome to .NET Framework/Core/Standard Hell!?

To make all of this work, you have to understand the target frameworks that you should be using so that you can reuse the same code across Windows and Linux. After all, that is the true power of .NET Core. Unfortunately, all of this is pretty confusing. It is hard to grasp the different between .NET Framework, .NETCoreApp and .NETStandard. Hopefully, this article and example help make it less confusing and not more confusing.

The goal of .NETStandard is to solve the exact problem we are highlighting in this article. It provides a common surface area that can be targeted by different types of applications. You can create a class library and use it within an application deployed to Windows and Linux both. This gives you ultimate portability and reuse of your code.

If you really want to get into the mess of all this, you should try making a NuGet package that requires references to ASP.NET Core. You are forced to target .NETStandard 1.6 and .NET 4.5.1 both. You have to do this so people using ASP.NET Core, either way, can use your package. There is no way to target a single framework that works for both.

## “Self Hosting” ASP.NET Core in a Windows Service

You can absolutely “self-host” the Kestrel web server and ASP.NET Core in a Windows Service. This is similar to self-hosting ASP.NET Web API or WCF in a Windows Service. Although, using IIS as a reverse proxy in front of Kestrel is the preferred deployment option, especially for public facing web servers.

Self-hosting in a Windows Service is most likely used in software that is distributed. Prefix is a good example of this. We decided to make the UI all HTML based and we leverage an embedded web server within the app to power it. Self-hosting enables some interesting deployment scenarios without requiring the full installation of IIS on Windows desktops.

## Conclusion

The goal of this article was to showcase how to take a .NET Core project and deploy it as a Windows Service. To do so, you need to make sure your code is in a class library that targets .NETStandard. You can then use that class library in a Windows Service and in other .NET Core projects.

A Windows Service project must use the full .NET Framework. If you don't want to deploy any of your code to a Mac or Linux, you really don't need to use “.NET Core CLR” at all. You can target the full .NET Framework for all of your projects and still use .NET Core features like ASP.NET Core.

The real trick is mixing multiple projects that could be web apps, Windows apps, shared libraries, and non-Windows apps. To solve this problem you will want to use class libraries and .NETStandard so your code base can be shared.



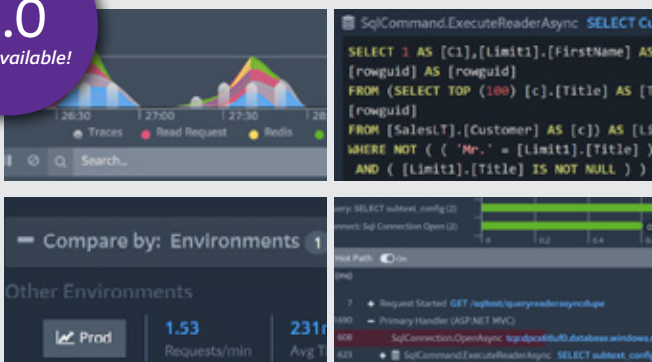
*“Prefix has helped me so much in writing better code. It's helped reduce calls to our database and to identify exactly why a few pages were taking so long to load. It is a great product and I am looking forward to seeing what Stackify comes up with next.”*

*–Patrick McCarthy, Andrews McMeel Universal*

**Prefix** Logs + Errors + Queries + More

3.0

Now Available!





# Master Logging for .NET Core

If you are getting started with ASP.NET Core, you are probably wondering what has changed with logging. In short, the answer is both nothing and everything. The common logging libraries you have always used with .NET still work. Microsoft does provide its own interface for logging with .NET Core and it uses it for .NET internals.

In this article, we will discuss using traditional .NET logging frameworks and some new changes in ASP.NET Core and .NET Core.

## Third Party Logging Libraries for ASP.NET Core

The three most popular libraries for logging support .NET Core. If you are converting your application from ASP.NET to Core, you won't have to make any code changes. Just upgrade to the latest NuGet packages. Although you may want to move your logging config to its own file since your web.config will now be gone in favor of the new appsetting.json.

### Log4Net

When .NET Core 1.0 launched, log4net had not been ported to support .NET Core (it has since). Their long delay in support for .NET Core is just one example of where log4net has been slow to keep up with new trends and features in logging frameworks. It may be the most popular framework, but NLog and Serilog are on the leading edge.

### NLog

NLog has quickly become the second most popular framework for .NET logging. They had support for .NET Core when v1.0 came out and continue to rapidly add new features. NLog even works across Xamarin, Mono, and other runtimes. NLog is a safe bet if you are thinking about selecting a new logging framework for ASP.NET Core.

## Serilog

Serilog was created to bring more structure to logging variables and objects. It has become very popular and continues to grow. It supports all the common logging features you would support, like configurable output targets and is much more modern than log4net.

## Built-in ASP.NET Core Logging

ASP.NET Core now has a built-in logging framework that you can use. It is not as feature-rich as third party libraries. Here's a quick tour of the new ILoggerFactory that is built into .NET Core.

If you have created a brand new ASP.NET Core web application, you have likely seen these loggerFactory lines of code in your Startup class.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();
    //removed the rest of the lines
}
```

You will also find some logging settings in the default appsettings.json.

```
{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  }
}
```

If you run the default app, you can see it write out some details in the debug window in Visual Studio that shows when MVC actions happen and how long they take.

```
Microsoft.AspNetCore.Hosting.Internal.WebHost: Information: Request starting
HTTP/1.1 GET http://localhost:29564/
Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker: Information:
Executing action method ASPNetCore.Controllers.HomeController.Index
(CoreSharedLib) with arguments () - ModelState is Valid
Microsoft.AspNetCore.Mvc.ViewFeatures.Internal.ViewResultExecutor: Information:
Executing ViewResult, running view at path /Views/Home/Index.cshtml.
Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker: Information:
Executed action ASPNetCore.Controllers.HomeController.Index (CoreSharedLib) in
26.8787ms
Microsoft.AspNetCore.Hosting.Internal.WebHost: Information: Request finished in
60.9946ms 200 text/html; charset=utf-8
```

## Disabling or Reducing the Built-in .NET Core Logging Messages

If you want to disable these built-in messages all together, you can simply remove the `AddConsole()` and `AddDebug()` methods. If you want to reduce how much they log then you can adjust their log levels.

For `AddConsole()` you will notice that you can pass into it `Configuration.GetSection("Logging")` which pulls the log levels from your `appsettings.json` configuration, or other custom providers you may be using. By setting a log level to "None" it will disable it. Just above is an example of an `appsettings.json`.

`AddDebug()`, for some reason, does not support the same configuration-based initialization. You can instead specify a log level in your code directly, which is probably fine since debug is really just meant for use in development and wouldn't need to be configured differently in QA or production.

These log levels are all supported: None, Trace, Debug, Information, Warning, Error, Critical.

## How to Redirect Internal .NET Core Logging to NLog or Serilog

You don't have to use the built-in logging. You can – and we would probably recommend this – use NLog, Serilog, or log4net. If you want to capture the internal logging coming from .NET Core and send it to your separate logging framework, you can. You just have to configure it!

For NLog/Serilog, all you have to do is add one line of code in your `Startup Configure()` method. This adds a special logging provider for NLog/Serilog that redirects the messages.

```
loggerFactory.AddNLog();
or
loggerFactory.AddSerilog();
```

## Using the New ILogger and ILoggerFactory in Your Code

Being able to see all of the .NET Core internal logging is cool, but what about your own logging?

One of the big additions to .NET Core is built-in dependency injection. You can use the ILoggerFactory via dependency injection in your own code as shown below.

```
public class HomeController : Controller
{
    ILoggerFactory _loggerFactory;
    public HomeController(ILoggerFactory loggerFactory)
    {
        _loggerFactory = loggerFactory;
    }
    public string Ping()
    {
        var logger = _loggerFactory.CreateLogger("LoggerCategory");
        logger.LogInformation("Calling the ping action");
        return "pong";
    }
}
```

Now, when we execute this code we see similar logging as I before, but we also see our extra logging line.

```
LoggerCategory: Information: Calling the ping action
```

So, you can see how easy it is to use the new built-in ASP.NET Core logging via ILoggerFactory and dependency injection.

The new logging API currently supports the following built-in providers:

- Console
- Debug
- EventSource
- EventLog
- TraceSource
- Azure App Service

So if you have followed along this far, you may be wondering about how to make the built-in logging write to a file on disk. In short, you have to use a 3rd party extension to do it.

## How to Enable ASP.NET Core Logging to a File

Since .NET Core does not support logging to file directly, you will want to use an extension from NLog or Serilog that can route .NET Core logging to those libraries. Then you can take full advantage of their features. We are going to assume that they left a File writer option out because of the complexity it creates. It opens up a huge set of requirements around max file sizes, rolling files, deleting files, file writing performance, etc.

Using Serilog as an example, all you have to do is install their Extensions NuGet package and, with one line of code, do an AddFile in your Startup class. Now you have a log file on disk!

```
public void Configure(IApplicationBuilder app,
                     IHostingEnvironment env,
                     ILoggerFactory loggerFactory)
{
    loggerFactory.AddFile("Logs/mylog-{Date}.txt");
    //other code removed for example
}
```

## Limitations of ASP.NET Core Logging Configuration

The built-in logging doesn't provide much for configuration. All you can do is configure the logging level (Debug, Warning, Error, etc) by logging category name.

## Viewing Your Application Errors and Logs

The screenshot shows the ASP.NET Logging Viewer for .NET Core. It displays a log of a request with the following details:

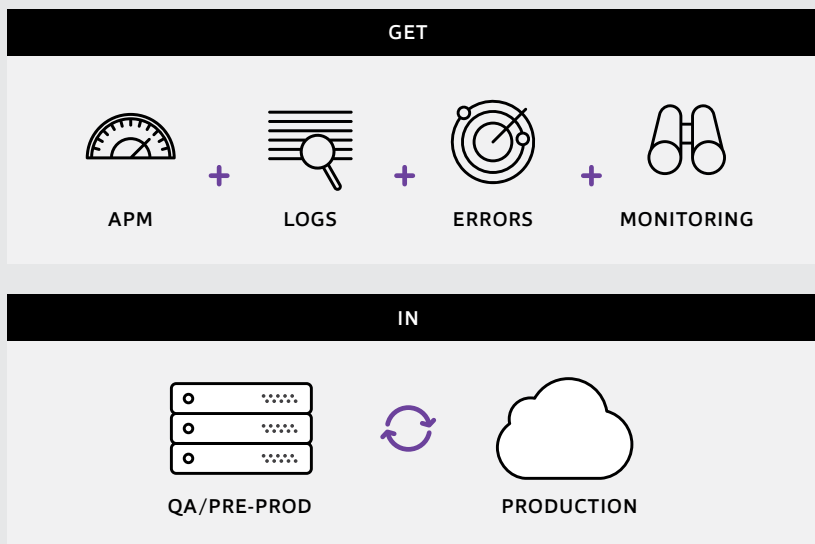
- Request details:** GET /stackify/devices
- Logging:** DEBUG Setting #redis ("key":"LastGetTopPeopleCalled")
- Redis cache call:** RedisClientAdd stackifydevcentral.redis.cache.windows.net|LastGetTopPeopleCalled
- Error in code:** Caught: System.InvalidOperationException: This error gets thrown all the time and thrown away causing CPU problems at HelloStackify.Web.Controllers.StackifyController.Devices
- SQL queries:** SELECT TOP (100) [c].[ProductCategoryID] AS [ProductCategoryID], [c].[ParentProductCategoryID] AS [ParentProductCategoryID], [c].[Name] AS [Name], [c].[rowguid] AS [rowguid], [c].[ModifiedDate] AS [ModifiedDate] FROM [SalesLT].[ProductCategory] AS [c]
- Web request status:** Request Completed Status: 200

## How to View Your Logs While Writing and Testing Code

You can use Prefix, Stackify's free ASP.NET Profiler, to view all of your logging statements during development. Prefix can show you all of your logs, errors, SQL queries, HTTP calls, and much more.

## How to View Your Logs in QA or Production

If you only send your logging to a file on disk, it can be pretty hard to get a lot of use out of them once you deploy your app to a server somewhere. You really need to centralize your logs with a log management system. This gives you the ability to do full-text searching across all of your logs for all applications and servers from one place.



*Do you see what your code is doing and why?*



*Is your dev team ready for better insights? Retrace is free for 14-days, then starts at only \$10/month.*

*Schedule a demo with our product experts now:*

*"Retrace quickly and easily finds the root cause of the issue. We saw a crash report in production and we resolved it right away. By the time a ticket came in, we were ready to push the fix. It's very good to be proactive on these errors."*

*We love the tool very much. It's saved us and our apps many times. It really helps get the error and logging information and helps make exception resolution turnaround quicker."*

*—Syad & Gajjar, Paycor*

# How to Use LoggerFactory and Microsoft.Extensions.Logging for .NET Core Logging with C#

If you have used .NET Core, you have probably battled with the new built-in .NET Core LoggerFactory which is in Microsoft.Extensions.Logging. It has created a lot of confusion around logging with ASP.NET Core. At Stackify, we have logging integrations for log4net, NLog, Serilog, and our direct API. We have a lot of experience with building logging appenders and libraries to work with various logging frameworks. As we have been working on perfecting our .NET Core support, we wanted to share some of our tips and findings.

## Basics of the .NET Core Logging With LoggerFactory

It is designed as a logging API that developers can use to capture built-in ASP.NET logging as well as for their own custom logging. The logging API supports multiple output providers and is extensible to potentially be able to send your application logging anywhere.

Other logging frameworks like NLog and Serilog have even written providers for it. So you can use the ILoggerFactory and it ends up working sort of like Common.Logging does as a facade above an actual logging library. By using it in this way, it also allows you to leverage all of the power of a library like NLog to overcome any limitations the built-in Microsoft.Extensions.Logging API may have. The biggest of those is being able to actually write your logs to a file on disk!

## Where is the LoggerFactory Created?

For an ASP.NET Core, you first see the ILoggerFactory within your Startup class that looks something like this below. This is where you can add providers to the ILoggerFactory for things like Debug & Console but also more advanced things like ETW or 3rd party providers.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
    ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging")); //log levels set
in your configuration
    loggerFactory.AddDebug(); //does all log levels
    //removed rest of code
}
```

If you are like us, you immediately want to know what calls Configure() and where the ILoggerFactory instance actually came from and live at.

When your app starts up you use the WebHostBuilder to essentially start up Kestrel and your web server.

```
public static void Main(string[] args)
{
    var host = new WebHostBuilder()
        .UseKestrel()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .UseIISIntegration()
        .UseStartup()
        .Build();

    host.Run();
}
```

This process under the covers ultimately calls some code in the WebHostBuilder class that creates a new LoggerFactory() and stores the reference to it in the ServicesCollection and the WebHostBuilder itself.

```
private IServiceCollection BuildHostingServices()
{
    //removed code
    ServiceCollection services = new ServiceCollection();
    services.AddSingleton(this._hostingEnvironment);
    if (this._loggerFactory == null)
        this._loggerFactory = (ILoggerFactory) new LoggerFactory();

    using (List<Action>.Enumerator enumerator = this._configureLoggingDelegates.
        GetEnumerator())
    {
        while (enumerator.MoveNext())
            enumerator.Current(this._loggerFactory);
    }
    services.AddSingleton(this._loggerFactory);
    services.AddLogging();
    //removed code
}
```

## Accessing the LoggerFactory Object via Dependency Injection and Services

In the example code below, we are showing off 3 different ways to access the LoggerFactory from your MVC controller. Dependency injection can give you the factory or a logger either one. Also, because it is stored in the services collection that later gets set on each HttpContext, you can potentially access it that way.

```
public class ValuesController : Controller
{
    private ILoggerFactory _Factory;
    private ILogger _Logger;

    //set by dependency injection
    public ValuesController(ILoggerFactory factory, ILogger logger)
    {
        _Factory = factory;
        _Logger = logger;
    }

    [HttpGet]
    public IEnumerable Get()
    {
        var loggerFromDI = _Factory.CreateLogger("Values");
        var loggerFactory = this.HttpContext.RequestServices.GetService();
        var loggerFromServices = loggerFactory.CreateLogger("Values");

        _Logger.LogDebug("From direct dependency injection");
        loggerFromDI.LogDebug("From dependency injection factory");
        loggerFromServices.LogDebug("From services");
    }
}
```

## Accessing the Logging API Outside of a MVC Controller

This is where the new logging API quickly becomes a nightmare. Dependency injection works great for accessing it in your MVC controller. But...how do you do logging in a class library that is consumed by this MVC project?

1. You could pass your existing LoggerFactory into every object/method you call (which is a terrible idea).
2. You could create your own LoggerFactory within your class library. This is an option as long as you don't use any providers like a File provider that can't have multiple instances writing

at the same time. If you are using a lot of different class libraries you would have a lot of LoggerFactory objects running around.

3. Create a centrally located static class or project to hold and wrap around the main LoggerFactory reference

We see number three as the best solution here, unless you aren't using any providers that have concurrency issues.

## How to Use the Logging API from Everywhere

Our suggestion is to create a little static helper class that becomes the owner of the LoggerFactory. The class can look something like this below. You can then use this ApplicationLogging class in any code that you want to use logging from and not have to worry about recreating LoggerFactory objects over and over. After all, logging needs to be fast!

```
public class ApplicationLogging
{
    private static ILoggerFactory _Factory = null;

    public static void ConfigureLogger(ILoggerFactory factory)
    {
        factory.AddDebug(LogLevel.None).AddStackify();
        factory.AddFile("logFileFromHelper.log"); //serilog file extension
    }

    public static ILoggerFactory LoggerFactory
    {
        get
        {
            if (_Factory == null)
            {
                _Factory = new LoggerFactory();
                ConfigureLogger(_Factory);
            }
            return _Factory;
        }
        set { _Factory = value; }
    }

    public static ILogger CreateLogger() => LoggerFactory.CreateLogger();
}
```

## How to Capture ASP.NET Built-In Logging and Your Logging

There is one really key thing you need to know if you want to capture the built-in ASP.NET logging messages:

ASP.NET will only write its internal logging to the `LoggerFactory` object that it creates at app startup. So, if you new up (`new LoggerFactory()`) on your own anywhere else in your code, ASP.NET is not going to write its logging to it. This even includes my recommendation around this `ApplicationLogging` class.

What you need to do is, in `Startup.Configure`, grab that reference of the `LoggerFactory` and set it to your static logging class so it becomes your primary reference.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
    ILoggerFactory loggerFactory)
{
    //call ConfigureLogger in a centralized place in the code
    ApplicationLogging.ConfigureLogger(loggerFactory);
    //set it as the primary LoggerFactory to use everywhere
    ApplicationLogging.LoggerFactory = loggerFactory;
    //other code removed
}
```

If you follow this pattern you will only have one `LoggerFactory` object in your application, and you can access it anywhere via your `ApplicationLogging` static helper class. We wish the .NET team would make this built-in to the logging API somehow to simplify this.

## Extend the Microsoft.Extensions.Logging API Functionality by Using NLog or Serilog Providers

Both NLog and Serilog have a provider that you can use to extend the functionality of the built-in logging API. They essentially redirect all of the logs being written to the new logging API to their libraries. This gives you all the power of their libraries for the output of your logs while your code is not tied to any particular library. This is similar to the benefit of using `Common.Logging`.

Another option is to just use NLog or Serilog and don't even worry about the new logging API. If you want to capture the built-in ASP.NET logging, you can plugin the NLog/Serilog provider and it will map those messages over. By doing it this way, you can use a different logging library directly and you don't have to even think about `LoggerFactory` even existing.

**Our suggestion: Just keep doing logging how you always have.**

After all the time we've spent messing with the new `Microsoft.Extensions.Logging` API, we suggest using NLog and not even worrying about the new `LoggerFactory`.

# Deploy to and Optimize .NET Core Web Apps in Azure

ASP.NET Core is the latest version of ASP.NET. It is built from the ground up using .NET Core, the lightweight, composable new .NET Framework. Some people love it, some don't. However, this is the future. Microsoft will move forward with .NET Core and in the future, the regular, full-fledged versions will go away.

This article will show you how to deploy a simple ASP.NET Core application to Azure. We'll also show you how to instrument the application with Stackify Retrace, which is an Application Performance Management tool.

We will use an example application called SimplCommerce. This is an open source e-commerce application, based on ASP.NET Core, hosted on GitHub. Basically, the application consists of a web application and a database.

The steps we will follow are:

1. Create the SimplCommerce Database on SQL Azure
2. Configuring the Application to Use Stackify Retrace
3. Deploying the Application to Azure
4. Activating Stackify Retrace for the Web App
5. Stackify Retrace Benefits in Action

## Create the SimplCommerce Database on SQL Azure

The SimplCommerce application uses a SQL database to store its product catalog and other data. To start using the application, you first need to setup a database. You can do this on your local machine, or somewhere else.

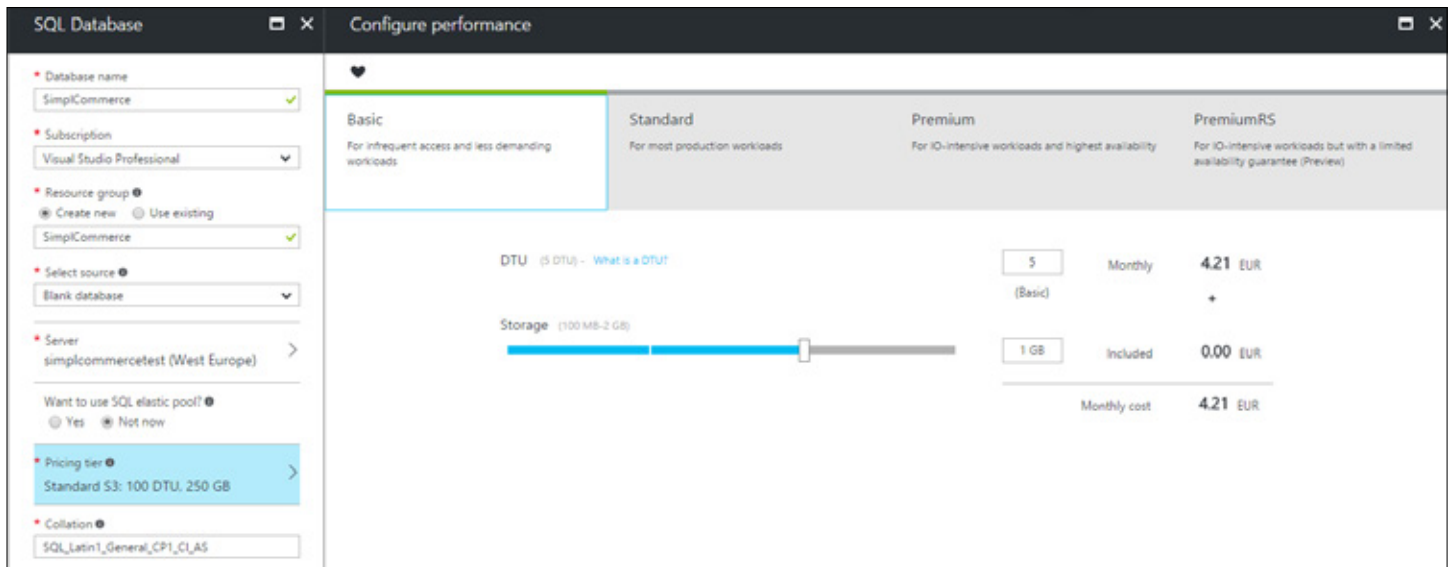
Our plan is to deploy the whole application to Azure. When working in Azure, we like to work with a database that runs there as well, as that provides confidence that whatever we're doing runs in Azure SQL and we're not going to run into any weird differences with an on-premises SQL Server.

To setup up an Azure SQL Database and connect to it, we need to execute the following steps:

1. Create an Azure SQL Database
2. Change the connectionstring of the application
3. Open the firewall to allow my local machine to connect to the Azure SQL Server

## Step 1: Create an Azure SQL Database

Step one is to create an Azure SQL Database to host the Simplicommerce data. We do this by creating new database in the Azure Portal. This takes us through a wizard that lets us create a database, an accompanying Azure SQL Server and configure them.



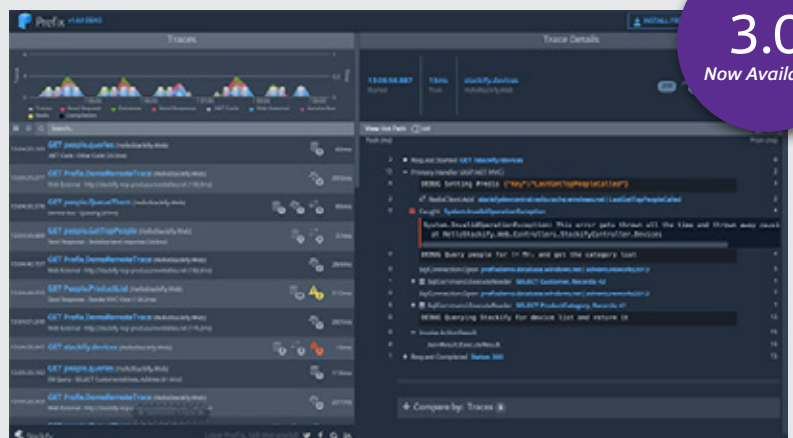
As you can see in the image above, we chose the basic pricing tier, as this is just a test application and won't need a lot of horsepower.

That's it, Azure SQL server and database created. Now to connect to it. We like to connect to the Azure SQL Database from a local machine and use it for local development. Although this seems slow because the database is somewhere else than our code, it really is fast enough to work with.

## Prefix

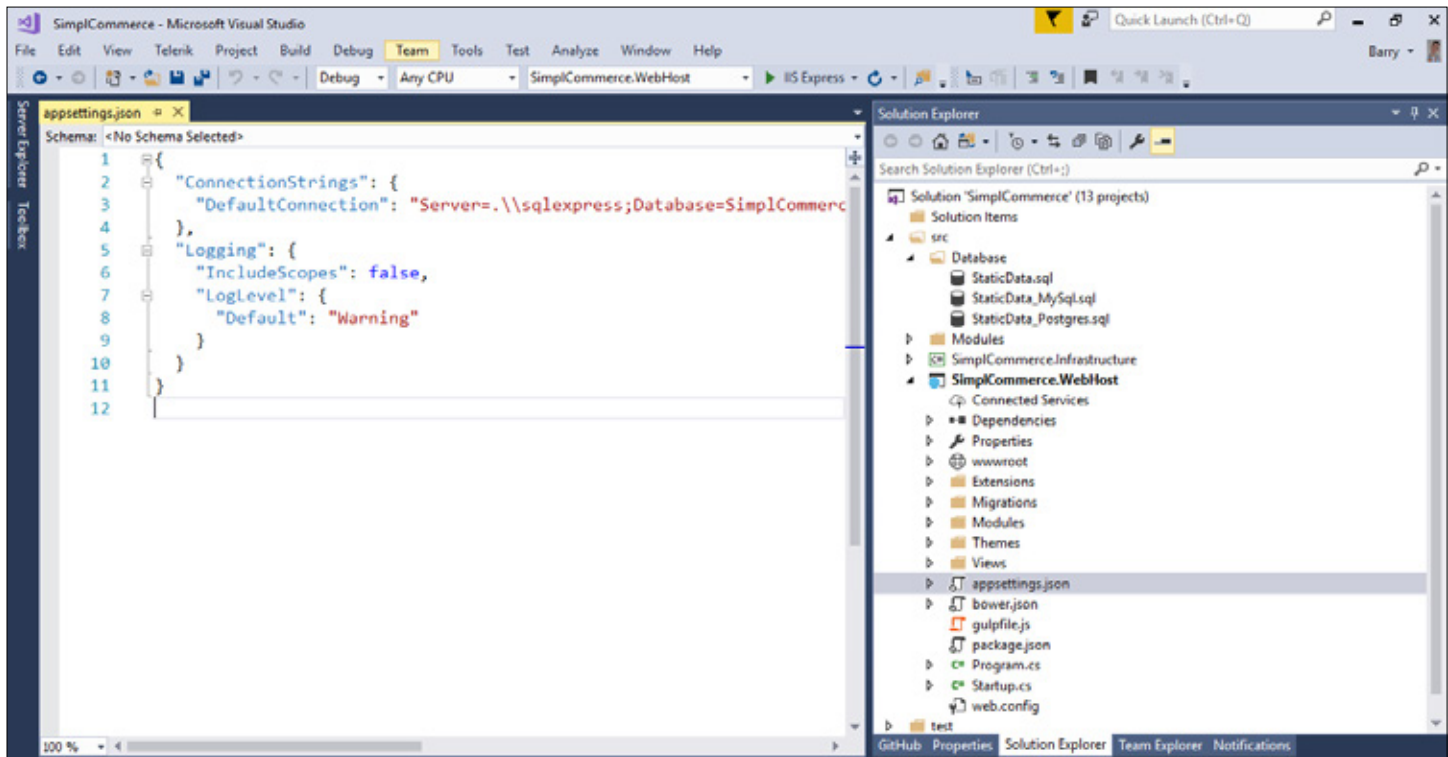
*Prefix is a lightweight tool that shows .NET developers:*

**Logs + Errors + Queries + More**  
*in real time—and it's free!*

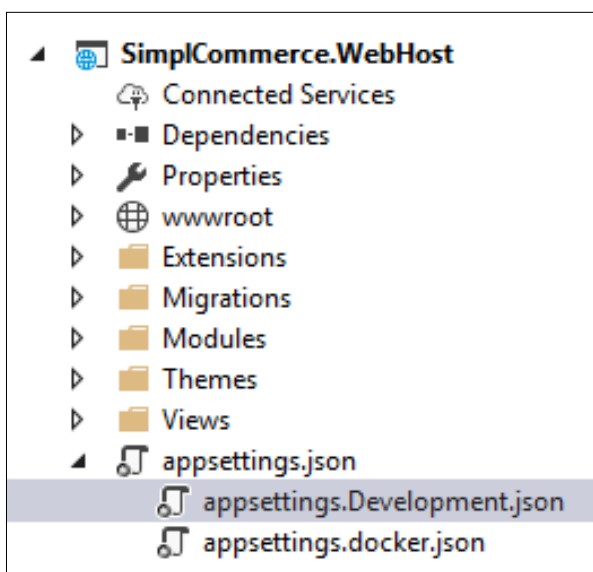


## Step 2: Change the connectionstring of the Application

To connect the SimplCommerce application to the database, we need to change its connectionstring. In ASP.NET Core, this is located in the appsettings.json file and underlying file(s). These are .json files, that contain configuration, just like the web.config did in the past.



The beauty of these is that they work based on the environment variable that you set. For instance, you can create an appsettings.development.json file for when the environment variable is development, like this:



The environment variable that we're talking about is the ASPNETCORE\_ENVIRONMENT variable. By default, when you debug in Visual Studio, ASP.NET Core uses this variable with the value 'development'. When you deploy your application, by default, the variable will have the value 'production'. You can overwrite this variable in whatever hosting environment you use or even in the settings of the projectfile in Visual Studio.

To connect to the database from our local machine, we have put the connectionstring to the database in the appsettings.development.json file.

## Step 3: Open the firewall to allow your local machine to connect to the Azure SQL Server

But that's not enough. By default, Azure SQL Server has a firewall enabled, blocking all IP-addresses, except addresses within Azure, from accessing the server. Therefore, we need to add our IP-address to the firewall, so that we can reach the server. You can do that from the Azure Portal, in the settings of the Azure SQL Server.

That should do it. However, the database is empty and doesn't even have any tables in it. On the [SimplCommerce GitHub page](#), you'll find the steps that you need to execute to fill the database with tables and data. This mainly involves running an Entity Framework migration and running a SQL script to fill the database with data.

## Configuring the Application to Use Stackify Retrace

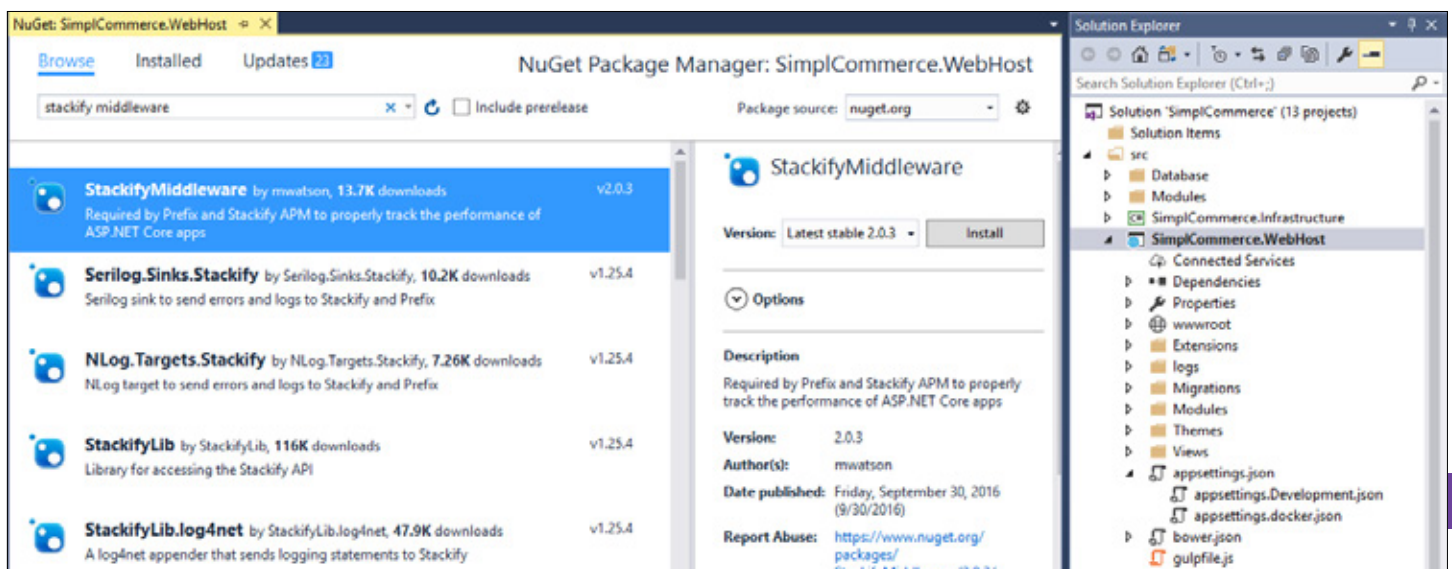
Before we deploy the application to Azure, we want to get it ready to send data to Stackify Retrace, so that we can monitor the app and see how it's doing. This is a very powerful monitoring tool that allows us to monitor things like:

- Performance of the application and dependencies
- Errors within the application
- Availability of the application

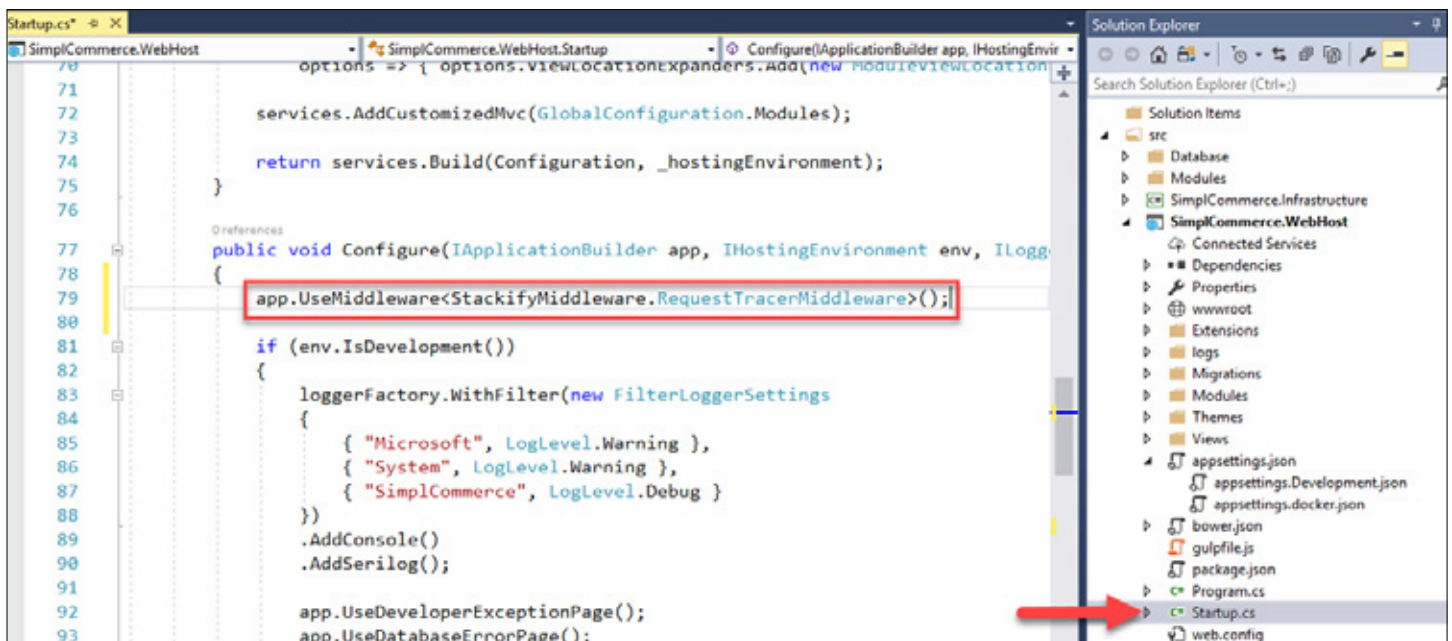
That first one is really important. We want to see the complete performance of the application. This means that we also want to see how our dependencies, like our Azure SQL Database, are performing and how we can improve that.

The ability to be alerted when an error occurs in the app and drill down into it is of vital importance. Without it, we wouldn't even know that errors are happening until we hear about it from a user, which is bad. Also, Retrace allows us to dive deeply into the error so that we can trace back what caused it and fix it.

All of this is really simple to set up. First, we add a nugget package called **StackifyMiddleware** which you can do in Visual Studio like in the image below or by typing the following into the package manager console window: **Install-Package StackifyMiddleware**



After that, you need to activate the Stackify middleware, which you can do in the class of the application. Here, in the Configure method, you need to type the line **app.UseMiddleware<StackifyMiddleware.RequestTracerMiddleware>()**; Be sure to insert this line before the app.UseMvc statement.



And that's it. Now, we are ready to deploy the application to Azure.

## Deploying ASP.NET Core to Azure

We want this application to run in Azure. To do that, we will create an Azure Web App and deploy the application to it. ASP.NET Core applications run in Azure Web Apps, just like any other applications. Bear in mind though that older ASP.NET Core applications (based on the project.json format) are not supported in Azure as of June 2017.

We will deploy the application to Azure using Visual Studio. This is really simple. You just right-click the project file of the ASP.NET App and choose publish. After that, a wizard like this appears that guides you through the process. It will even create a new Web App for you if you didn't have one already.

### Create App Service

Host your web and mobile applications, REST APIs, and more in Azure

Microsoft account  
b.luijbregts@waardedoor.nl

Hosting

Services

Web App Name

SimplCommerceWebHost

Change Type ▼

Subscription

Visual Studio Professional

Resource Group

SimplCommerce (westeurope)

New...

App Service Plan

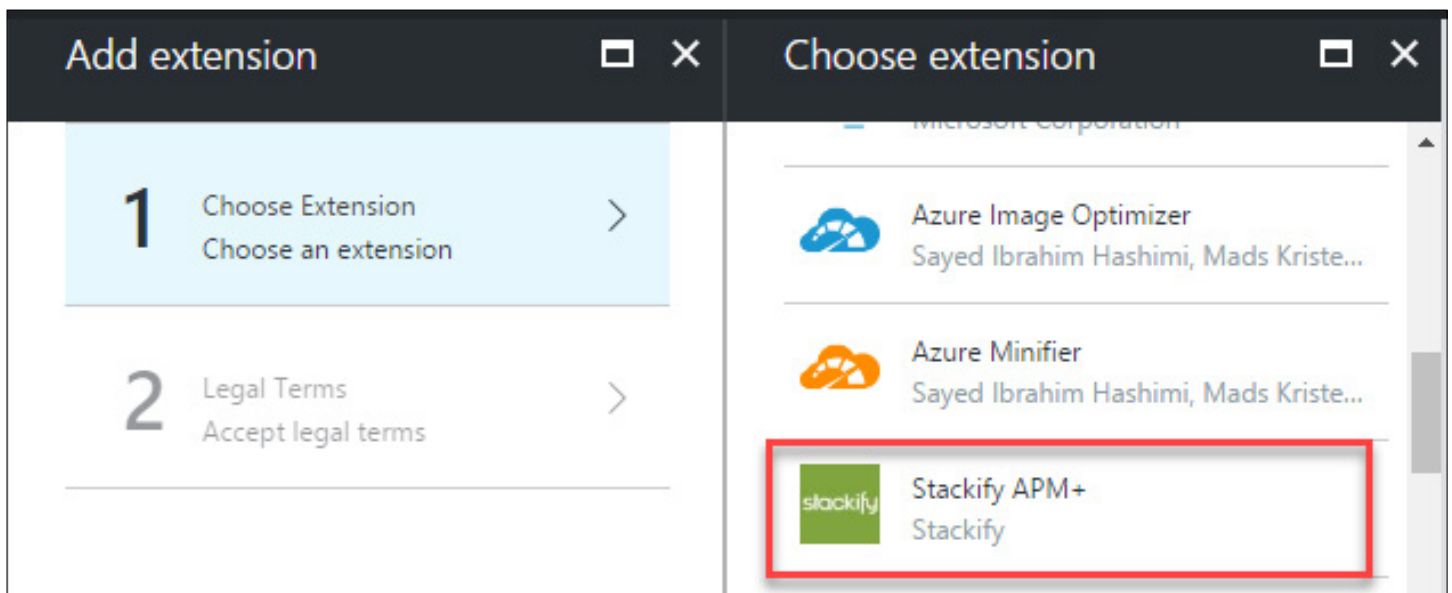
SimplCommerceWebHostPlan\*

New...

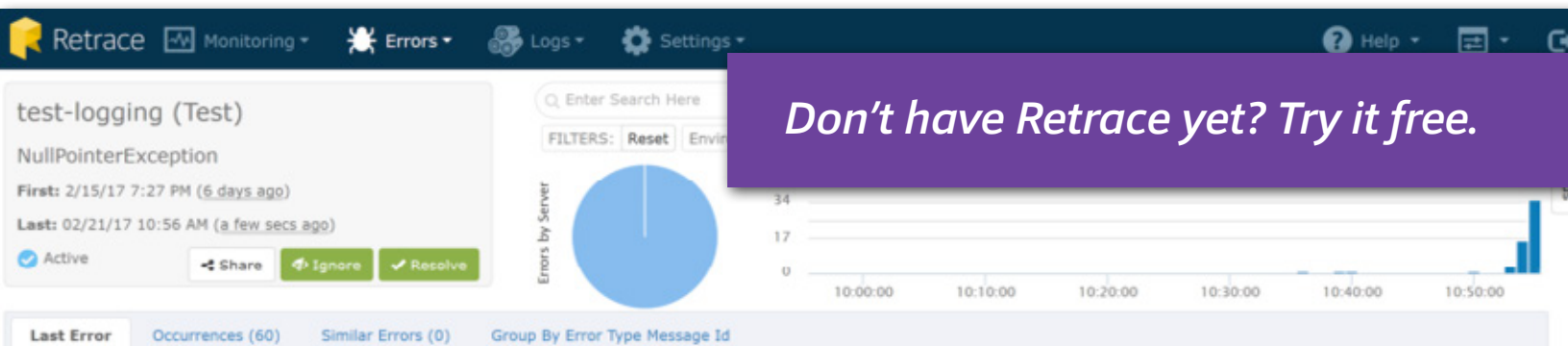
After this has completed, we have a Web App, with the application running in it. But it is not working yet.

## Activating Stackify Retrace for the Web App

First of all, we need to install an extension in order for Stackify Retrace to capture the data of the Web App. We can do this from the Azure Portal. Here, you can choose the extensions menu from the options of the Web App and choose the Stackify APM+ extension and install it. This will enable Stackify to capture metrics of the resources that run the Web App, like CPU, Memory and so on.



Make sure to restart the Web App after installing this extension.



*Retrace is a fully integrated, multi-environment app performance super-power with:*

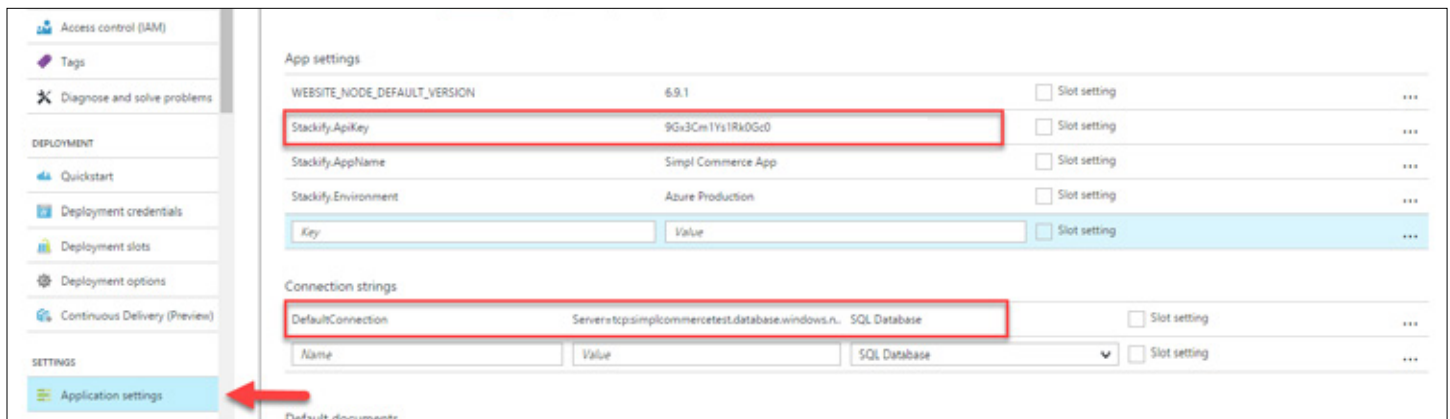
**APM + Errors + Logs + Metrics + Monitoring**

*Starting at just \$10/server/month, any team can afford to use it in pre-prod, production, at scale, whatever!*

# Configure the Web App

The next thing that we need to do is to configure the Web App.

Remember that we've put the connectionstring to the database in the **appsettings.development.json file**? Because of that, the application can't connect to the database, because by default, it doesn't use the **.development.json** file when it is deployed. Therefore, we need to put a connectionstring value in the ApplicationSettings of the Web App. This will override the configuration setting for the database connection, allowing the application to connect to the database.

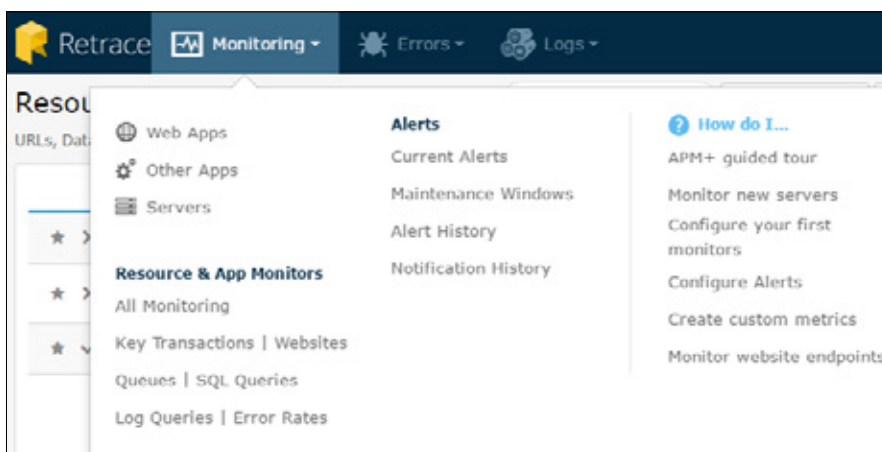


Also, we need to insert the **Stackify.ApiKey** and its value into the application settings. This will tell Stackify that this Web App belongs to our account and allows us to see the data for the app. You can find the value of the key in your [Stackify account panel](#).

We also recommended setting up the **Stackify.AppName** and **Stackify.Environment** properties. These will show up in the Stackify portal and will help you to identify your application.

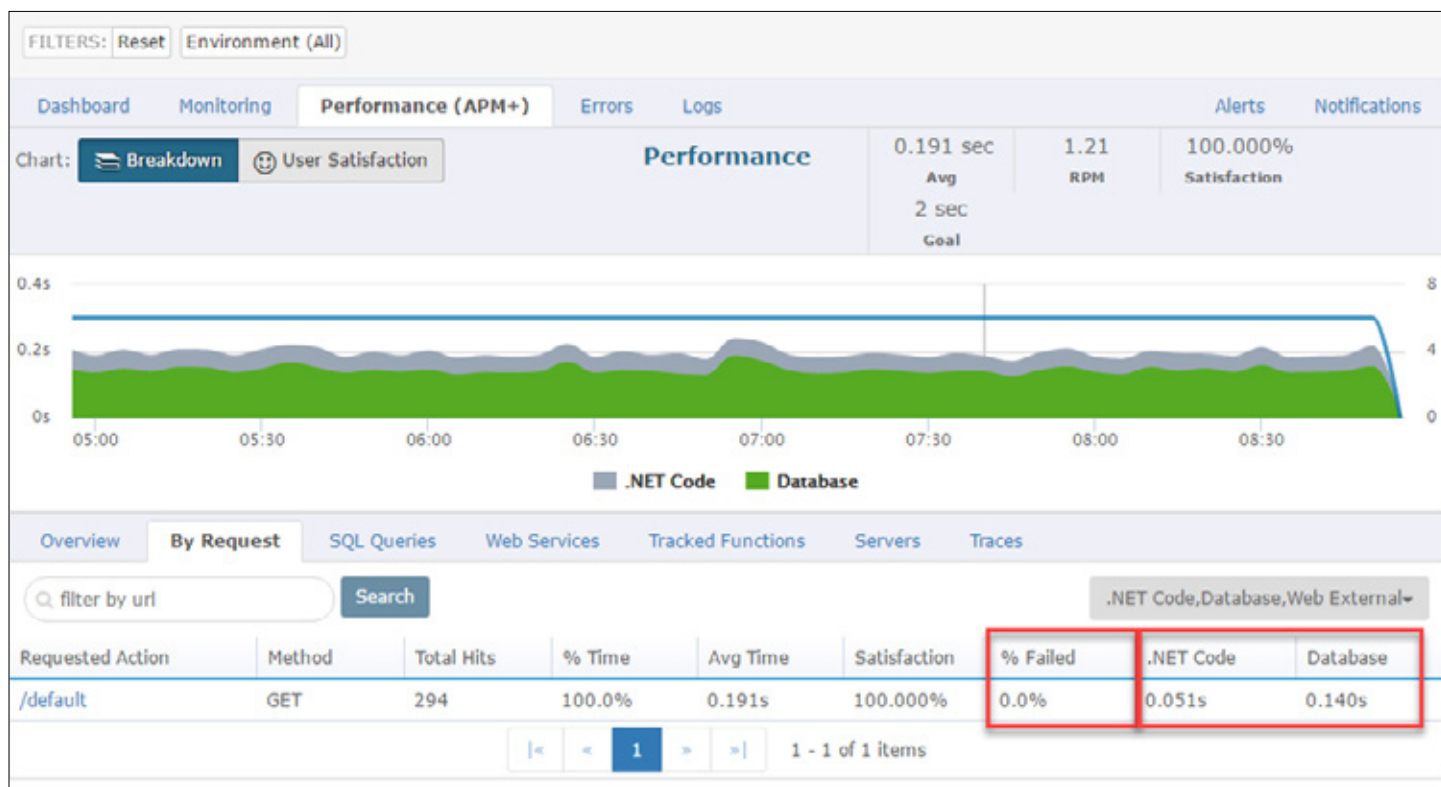
You can read the full instruction on setting up Stackify Retrace in an Azure App Services [here](#).

## Stackify Retrace Benefits in Action

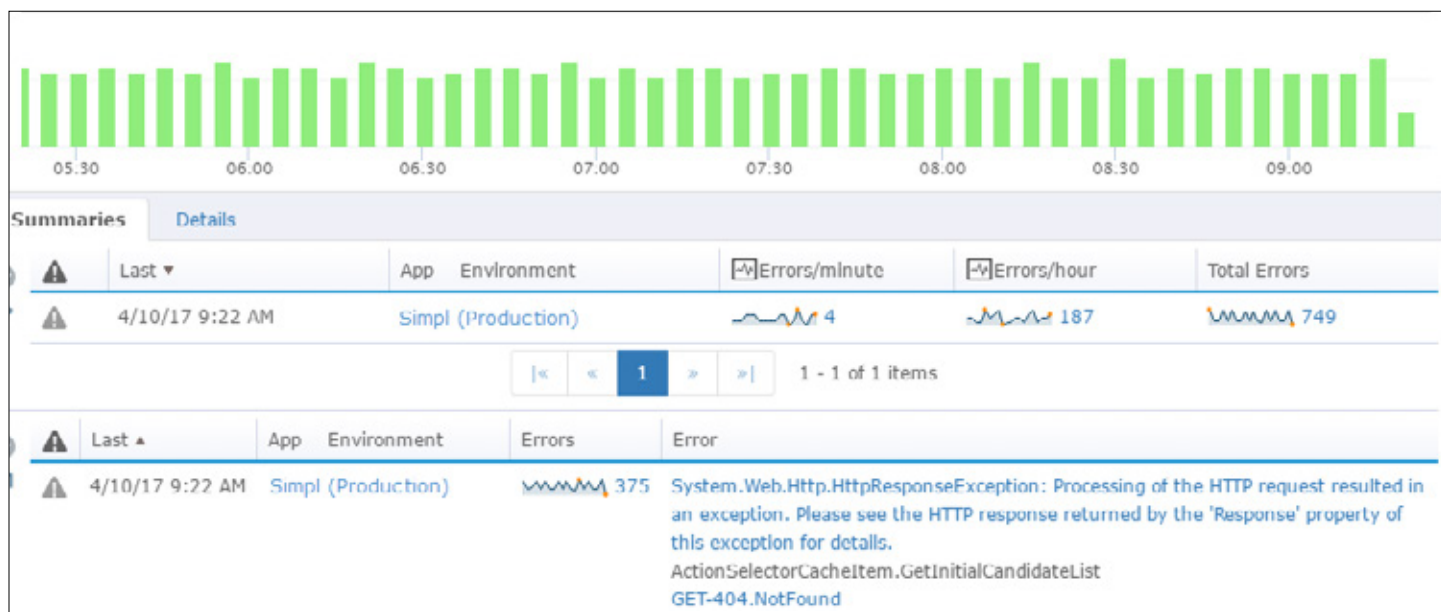


With Stackify Retrace running in the background, we can see some interesting information about the application. Stackify is a SaaS product, so you can see your data when you log in to <https://stackify.com>. Here, we have access to all sorts of dashboards and metrics, which we can customize to our heart's content.

One of Retrace's most popular dashboards is the Performance dashboard. Here, we can see very easily what the performance of our app is and where our performance comes from. Retrace can tell what our dependencies are, and look into those. In this case, we're calling into an Azure SQL Database, which, according to the graph, takes the most time. We can also see very quickly how many failed requests there are in the current time period.



Next to that, the error overview is very helpful. We've introduced an error in the application and let it run for a while to be able to show you this overview. This shows the amount of errors that are happening and what they are, including a short stacktrace.



You can drill down into the stacktrace to see everything about that error, including everything about the HTTP request, like the HTTP headers.

Error Detail

Details

Logs

Share

### Information

Occurred:	April 10, 2017 9:22 AM <b>ERROR</b> via StackifyLib-APM	
App:	Simpl Commerce (Production)	<a href="#">App Dashboard</a>
Server:	stackifywebapp2 demo dual [8069-2dc834] (D:\home\site\wwwroot)	<a href="#">Server Dashboard</a>
Url:	GET http://stackifywebapp2.azurewebsites.net/api/values/5	
Route:	values.5	
Method:	ActionSelectorCacheItem.GetInitialCandidateList	

### Error

System.Web.Http.HttpResponseException: Processing of the HTTP request resulted in an exception. Please see the HTTP response returned by the 'Response' property of this exception for details.

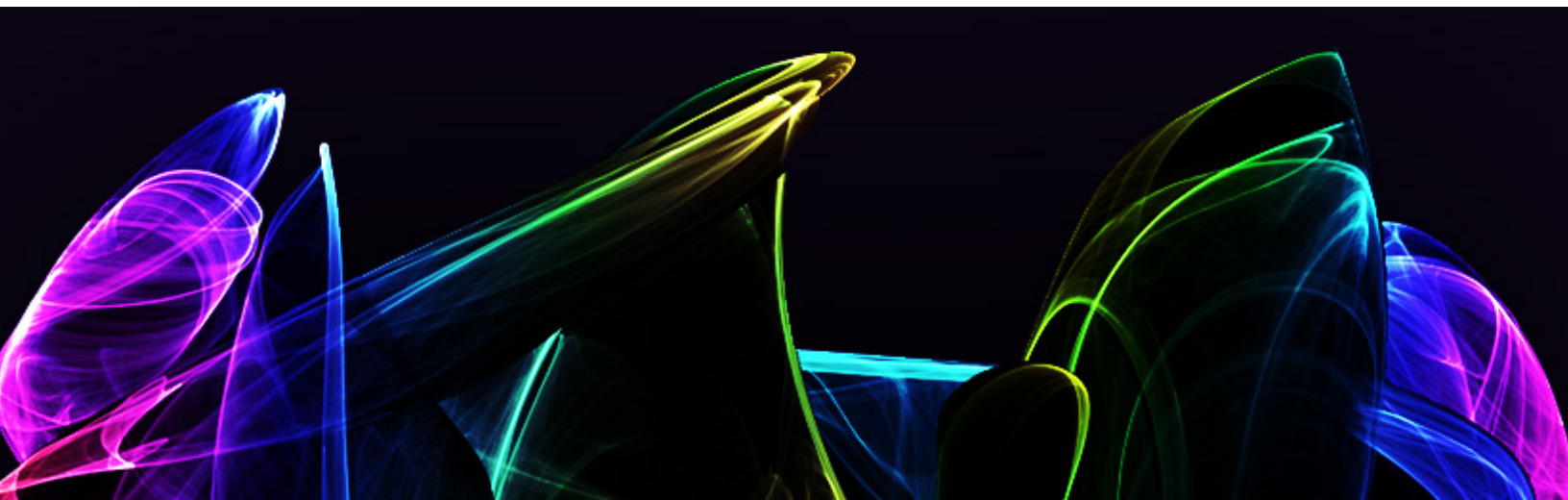
```
at ActionSelectorCacheItem.GetInitialCandidateList
at ActionSelectorCacheItem.FindMatchingActions
at ActionSelectorCacheItem.SelectAction
at System.Web.Http.Controllers.ApiControllerActionSelector.SelectAction
at System.Web.Http.ApiController.ExecuteAsync
at <SendAsync>d__1.MoveNext
```

### HTTP Request

```
GET http://stackifywebapp2.azurewebsites.net/api/values/5
```

[View All Occurrences](#)[Close](#)

This is very powerful and allows you to get to the bottom of an error quickly so that you can resolve it.



# Deploy ASP.NET Core to IIS with .NET Core Hosting

## How to Configure Your ASP.NET Core App For IIS

Deploying an ASP.NET Core app to IIS isn't complicated, but ASP.NET Core hosting is a little different than ASP.NET.

The first thing you will notice when creating a new ASP.NET Core project is they are actually console applications. Your project now contains a Program.cs file just like a console app would have and it contains the following code:

```
public class Program
{
    public static void Main(string[] args)
    {
        var host = new WebHostBuilder()
            .UseKestrel()
            .UseContentRoot(Directory.GetCurrentDirectory())
            .UseIISIntegration()
            .UseStartup()
            .Build();

        host.Run();
    }
}
```

## What is the WebHostBuilder?

All ASP.NET Core applications require a **WebHost** object that essentially serves as the application and web server. **WebHostBuilder** is used to configure and create the WebHost. You will normally see `UseKestrel()` and `UseIISIntegration()` in the WebHostBuilder setup code.

What do these do?

- **UseKestrel()** – This registers the `IServer` interface for Kestrel as the server that will be used to host your application. In the future, there could be other options, including [WebListener](#) which will be Windows only.
- **UseIISIntegration()** – This tells ASP.NET that IIS will be working as a reverse proxy in front of Kestrel. This then specifies some settings around which port Kestrel should listen on, forwarding headers, and other details.

If you are planning to deploy your application to IIS, `UseIISIntegration()` is required

## What is AspNetCoreModule?

You may have noticed that ASP.NET Core projects create a `web.config` file. This is only used when deploying your application to IIS. It registers the `AspNetCoreModule` as an HTTP handler.

Default `web.config` for ASP.NET Core:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModule"
resourceType="Unspecified"/>
    </handlers>
    <aspNetCore processPath="%LAUNCHER_PATH%" arguments="%LAUNCHER_ARGS%"
stdoutLogEnabled="false" stdoutLogFile=".\logs\stdout" forwardWindowsAuthToken="
false"/>
  </system.webServer>
</configuration>
```

`AspNetCoreModule` handles all incoming traffic to IIS and acts as the reverse proxy that knows how to hand the traffic off to your ASP.NET Core application. You can view the source code of it on [GitHub](#). It also ensures that your web application is running. It is responsible for starting your process up.

# Install .NET Core Windows Server Hosting Bundle

Before you deploy your application, you need to install the .NET Core hosting bundle for IIS. This will install the .NET Core runtime, libraries, and the ASP.NET Core module for IIS.

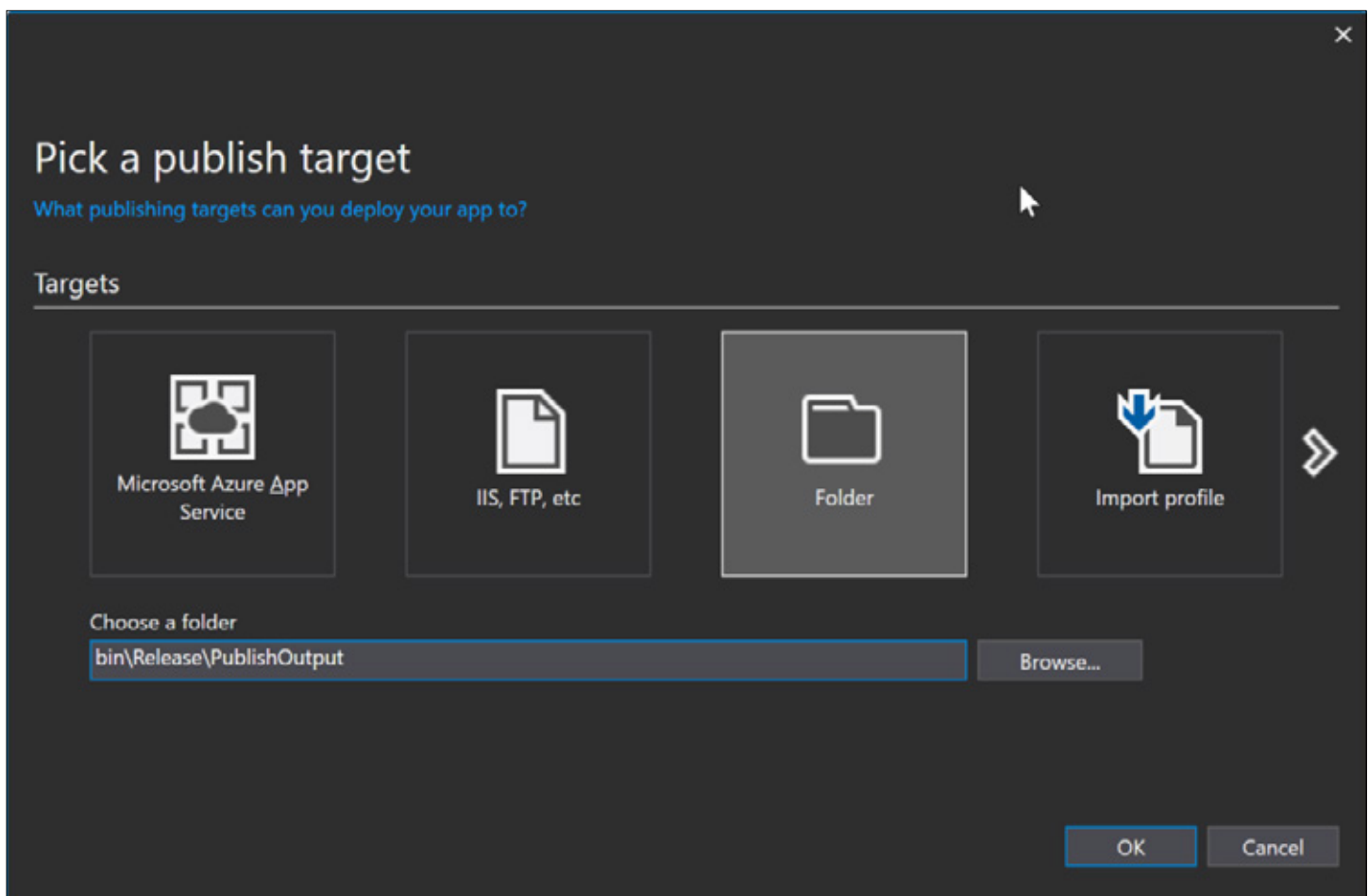
After installing it, you may need to do a "net stop was /y" and "net start w3svc" to ensure all the changes are picked up for IIS.

**Download:** [.NET Core Windows Server Hosting](#)

## Steps to Deploy ASP.NET Core to IIS

Before you deploy, you need to make sure that WebHostBuilder is configured properly to use Kestrel and IIS. Your web.config should also exist and look similar to our example above.

### Step 1: Publish to a File Folder



*Publish to folder with Visual Studio 2017*

## Step 2: Copy Files to Preferred IIS Location

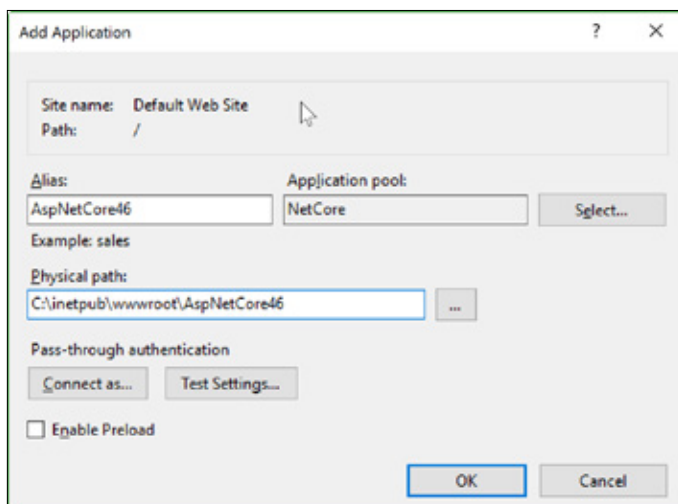
Now you need to copy your publish output to where you want the files to live. If you are deploying to a remote server, you may want to zip up the files and move to the server. If you are deploying to a local dev box, you can copy them locally.

For this example, we are copying the files to **C:\inetpub\wwwroot\AspNetCore46**

You will notice that with ASP.NET core there is no bin folder and it potentially copies over a ton of different .NET dlls. Your application may also be an EXE file if you are targeting the full .NET Framework. Our little sample project had over 100 dlls in the output.

<input type="checkbox"/> Name	Date modified	Type	Size
<input type="checkbox"/> refs	4/12/2017 12:54 PM	File folder	
<input type="checkbox"/> Views	4/12/2017 12:54 PM	File folder	
<input type="checkbox"/> wwwroot	4/12/2017 12:54 PM	File folder	
<input type="checkbox"/> appsettings.json	9/14/2016 2:13 PM	JSON File	1 KB
<input type="checkbox"/> ASPNetCore46.deps.json	4/12/2017 12:54 PM	JSON File	160 KB
<input checked="" type="checkbox"/> ASPNetCore46.exe	4/12/2017 12:54 PM	Application	14 KB
<input type="checkbox"/> ASPNetCore46.exe.config	4/12/2017 12:54 PM	XML Configuration Fi...	6 KB
<input type="checkbox"/> ASPNetCore46.pdb	4/12/2017 12:54 PM	Program Debug Data...	3 KB
<input type="checkbox"/> ASPNetCore46.runtimeconfig.json	4/12/2017 12:54 PM	JSON File	1 KB
<input type="checkbox"/> bower.json	9/14/2016 2:13 PM	JSON File	1 KB
<input type="checkbox"/> bundleconfig.json	9/14/2016 2:13 PM	JSON File	1 KB
<input type="checkbox"/> CoreSharedLib.dll	4/12/2017 12:53 PM	Application extension	35 KB
<input type="checkbox"/> CoreSharedLib.dll.config	12/21/2016 9:34 AM	XML Configuration Fi...	1 KB
<input type="checkbox"/> CoreSharedLib.pdb	4/12/2017 12:53 PM	Program Debug Data...	8 KB
<input type="checkbox"/> libuv.dll	6/13/2016 2:17 PM	Application extension	228 KB
<input type="checkbox"/> Microsoft.AspNetCore.Antiforgery.dll	2/17/2017 1:40 PM	Application extension	49 KB
<input type="checkbox"/> Microsoft.AspNetCore.Authorization.dll	2/17/2017 1:40 PM	Application extension	37 KB
<input type="checkbox"/> Microsoft.AspNetCore.Cors.dll	2/17/2017 1:40 PM	Application extension	31 KB

## Step 3: Create Application in IIS



The screenshot shows the 'Add Application' dialog box in IIS Manager. The 'Site name' is 'Default Web Site' and the 'Path' is '/'. The 'Alias' is 'AspNetCore46' and the 'Application pool' is 'NetCore'. The 'Physical path' is 'C:\inetpub\wwwroot\AspNetCore46'. The 'Pass-through authentication' section has 'Connect as...' and 'Test Settings...' buttons. The 'Enable Preload' checkbox is unchecked. The 'OK' and 'Cancel' buttons are at the bottom right.

First, create a new IIS Application Pool. You will want to create one under the .NET CLR version of "No Managed Code". Since IIS only works as a reverse proxy, it isn't actually executing any .NET code.

Second, create your new application under your existing IIS Site, or create a new IIS site. Either way, you will want to pick your new IIS Application Pool and point it at the folder you copied your ASP.NET publish output files to.

## Step 4: Load Your App!

At this point, your application should load just fine. If it does not, check the output logging from it. Within your web.config file you define how IIS starts up your ASP.NET Core process. Enable output logging by setting **stdoutLogEnabled=true** and you may also want to change the log output location as configured in **stdoutLogFile**. Check out the example web.config before to see where they are set.

## Advantages of Using IIS with ASP.NET Core Hosting

Microsoft recommends using IIS with any public facing site for ASP.NET core hosting. IIS provides additional levels of configurability, management, security, logging, and many other things.

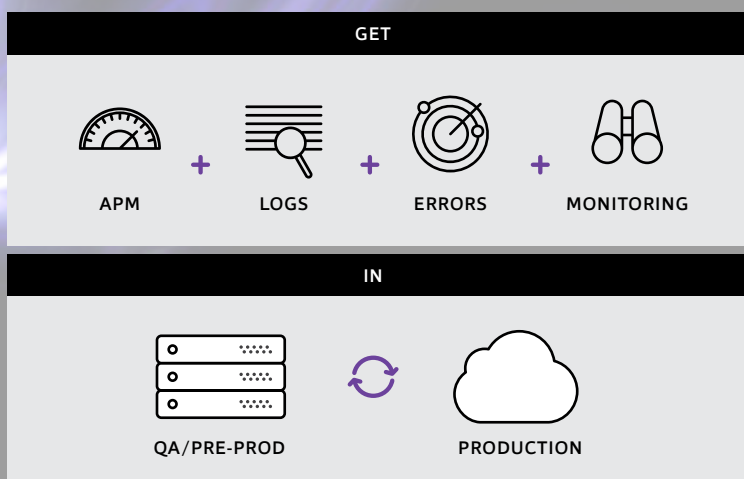
One of the big advantages to using IIS is the process management. IIS will automatically start your app and potentially restart it if a crash were to occur. If you were running your ASP.NET Core app as a Windows Service or console app, you would not have that safety net there to start and monitor the process for you.

*Retrace is one tool every developer team needs.*

*(And any developer team can afford)*

*"I've spent most of my career as a consultant at major corporations. Some of my smaller clients have asked that I take on operational roles, but it felt like there was too much—until I found Stackify. Now I can see trouble spots and drill in. Instead of having to spend all my time finding the hotspot, I can now spend that time fixing the hotspot."*

*—Steven Roberts, Broad Reach Software*



Retrace

STARTING AT ONLY

**\$10**/mo



# Lessons We Learned Converting from .NET Core

## Planning to Make the Switch?

At Stackify, we have been doing a lot of work with .NET Core since its release last year. We ported StackifyLib and our logging appenders over. We have also made sure that our app performance tools, Prefix and Retrace, can be used to profile .NET Core based apps. In fact, we've even converted the entire codebase of Prefix to core so it can run on a Mac! Below are some of the things we have learned to help convert ASP.Net to .Net Core.

### 1. Using xproj & csproj files together

There doesn't seem to be any way for these two project types to reference each other. You move everything to xproj, but then you can no longer use MSBuild. If you are like us, that means your current setup with your build server won't work. It is possible to use xproj and csproj files both at the same time which is ultimately what we ended up doing for our Windows targeted builds of Prefix.

### 2. Building for deployment

If you are planning to build an app that targets non Windows, you have to build it on the target platform. In other words, you can't build your app on Windows and then deploy it to a Mac. You can do that with a NetStandard library, but not a NetCoreApplication. They are hoping to remove this limitation in the future.

### 3. NetStandard vs NetCoreApplication1.0

What is the difference? NetStandard is designed as a common standard so that .NET 4.5, Core, UWP, Xamarin and everything else has a standard to target. So, if you are making a shared library that will be a nuget package, it should be based on NetStandard. Learn more about NetStandard here: <https://github.com/dotnet/corefx/blob/master/Documentation/architecture/net-platform-standard.md>

If you are making an actual application, you are supposed to target NetCoreApp1.0 as the framework if you plan on deploying it to Macs or Linux. If you are targeting Windows, you can also just target .NET 4.5.1 or later.

## 4. Almost no APM tool supports .NET Core. Almost.

One of the most requested features by New Relic users is support for .NET Core because Microsoft's newest version of the .NET framework that is leaner, faster, open source, and can even run on Linux.

As of January 2017, major APM providers like New Relic and AppDynamics do not support .NET Core—a full year after the initial betas of .NET Core were released. We would expect them to update their .NET docs when support is released. Requests for .NET Core is one of the most popular subjects in their customer forums.

The only alternative currently on the market is Retrace, which not only competes head-to-head with the other major APM options, but also includes errors, loggings, metrics, and monitoring in the same tool (not to mention being a fraction of the price).

Retrace works perfectly with:

- **.NET Core web apps on Windows:** Apps targeting the full .NET framework, as well as Core CLR/netcoreapp, are supported. Retrace works with ASP.NET Core apps running behind IIS or running kestrel directly.
- **.NET Core service apps on Windows:** Apps targeting the full .NET framework, as well as Core CLR/netcoreapp, are supported. If you have a .NET Core windows service or console application, you can configure Stackify Retrace to profile and monitor these applications. You must tell Retrace to specifically monitor non-web apps.

Unlike New Relic and most other APM tools, Stackify's entire platform is written in .NET. Because of that, we have fantastic support for ASP.NET, .NET, .NET Core, Microsoft Azure, and related technologies. Dozens of common application frameworks and dependencies are supported.

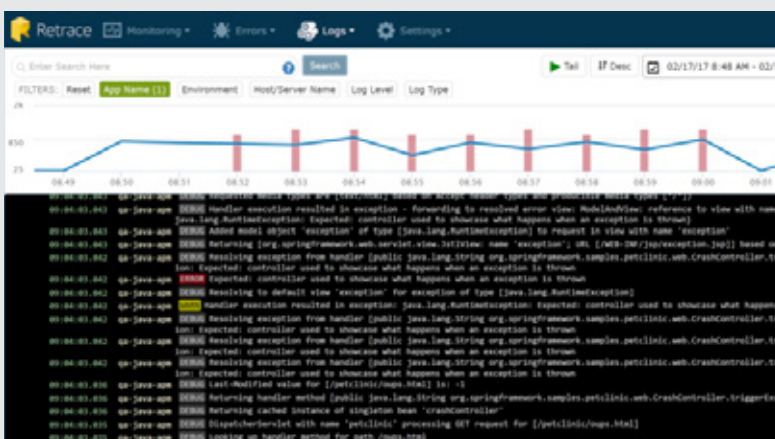
*"I am using Stackify to monitor a multi-tier application in the cloud, and was amazed at the easy set-up with no extra configuration."*

*Competitors seemed overly confusing, and costs would have exceeded our total hosting cost. Stackify was an easy choice. Honestly, we haven't experienced a single drawback."*

*—Aaron Mehlmauer, Management Science Associates*

## Retrace

**APM + Errors + Logs + Monitoring  
in one killer tool – QA & production!**



## 5. IIS is dead, well sort of.

As part of .NET Core, Microsoft (and the community) has created a whole new web server called Kestrel. The goal behind it has been to make it as lean, mean, and fast as possible. IIS is awesome but comes with a very dated pipeline model and carries a lot of bloat and weight with it. In some benchmarks, we have seen Kestrel handle up to 20x more requests per second. Yowzers!

Kestrel is essentially part of .NET Core which makes deploying your web app as easy as deploying any console app. As matter of fact, every app in .NET Core is essentially a console app. When your ASP.NET Core app starts up, it activates the Kestrel web server, sets up the HTTP bindings, and handles everything. This is similar to how self hosted Web Api projects worked with Owin.

IIS isn't actually dead. You can use IIS as a reverse proxy sitting in front of Kestrel to take advantage of some of it's features that Kestrel does not have. Things like virtual hosts, logging, security, etc. Microsoft still recommends using IIS to sit in front of your ASP.NET Core apps.

If you have ever made a self hosted web app in a Windows service or console app, it all works much differently now. You simply use Kestrel. All the self hosted packages for WebApi, SignalR and others are no longer needed. Every web app is basically self hosted now.

## 6. HttpModules and HttpHandlers are replaced by new "middleware."

Middleware has been designed to replace modules and handlers. It is similar to how Owin and other languages handle this sort of functionality. They are very easy to work with. Check out the ASP.NET docs to learn more. The good (and bad) news is you can't configure them in a config file either. They are all set in code.

## 7. FileStream moved to System.IO.FileSystem ???

Some basic classes that everyone uses on a daily basis have been moved around to different packages. Something as common as FileStream is no longer in the System.IO assembly reference/package. You now have to add the package System.IO.FileSystem. This is confusing because we are using class namespaces that don't directly match the packages.

## 8. StreamReader constructor no longer works with a file path.

Some simple uses of standard libraries have changed. A good example is the StreamReader which was often used by passing in a file path to the constructor. Now you have to pass in a stream. This will cause small refactorings to use a FileStream in addition to the StreamReader everywhere.

Another good example of this is around reflection. GetType() now returns a more simplified object for performance reasons and you must do a GetTypeInfo() to get the full details. Luckily that is backwards compatible to .NET 4.5.

## 9. Platform specific code... like Microsoft specific RSA

.NET Core is designed to run on Windows, Macs and Linux. But some of your code could potentially compile on Windows but then fail at runtime when you run it on a Mac or Linux. A good example of this is RSACryptoServiceProvider which appears to be useable. At runtime on a Mac, you will get a "platform not supported" type exception. Evidently this RSA provider API is Windows specific. Instead you have to use RSA.Create() which is a more generic implementation and has slightly different methods. Both are in System.Security.Cryptography. Confusing huh? The old "If it builds, ship it!" mentality totally falls apart on this one!

## 10. Newtonsoft changed to default to camel case on field names.

This has to be one of the biggest headaches of the conversion. Newtonsoft now defaults to camelCase. This will cause all sorts of REST APIs to break if you were using PascalCase. We ended up using the JsonProperty attribute on some things to force their casing how we needed them. This one is a big land mine, so watch out for it. #TeamPascalCase

## 11. System.Drawing doesn't exist.

Need to resize images? Not with the .NET framework currently.

There are some community projects that you can use. Check out Hanselman's blog post: Server-side Image and Graphics Processing with .NET Core and ASP.NET 5

## 13. DataSet and DataTable doesn't exist.

People still use these? Actually, some do. We have used DataTables for sending a table of data to a SQL stored procedure as an input parameter. Works like a charm.

## 14. Visual Studio Tooling

We have seen a lot of weirdness with intellisense and Visual Studio in general. Sometimes it highlights code like it is wrong, but it compiles just fine. Being able to switch between your framework targets is awesome for testing your code against each. Although we just removed net451 as a target framework from our project, but Visual Studio still thinks we are targeting it... There are still a few bugs to be worked out.

## 15. HttpWebRequest weird changes

In .NET 4.5 there are some properties you have to set on the HttpWebRequest object and you can't just set them in the headers. Well, in Core, they decided to reverse course and you have to use the header collection. This requires some hackery and compiler directives... Otherwise you get errors like this from your .NET 4.5 code: The 'User-Agent' header must be modified using the appropriate property or method. We need some extension methods for core to make it backwards compatible.

## 16. Creating a Windows Service in .NET Core

Windows Services can be easily created with Visual Studio 2017 as long as your code targets the full .NET Framework. You will need move your .NET Core code to a class library that targets NetStandard that can then be shared via other .NET Core apps and your Windows Service.

## 17. Web API is Gone/Part of MVC Now

With .NET Core Microsoft and the community decided to merge Web API and MVC together. They have always been very similar to work with and either could be used for API type applications. So in a lot of ways, merging them made sense.

## 18. Database access

Low level access via SqlConnection and SqlCommand works the same. My favorite two ORMs, Dapper and Entity Framework, both work with dotnet core. Entity Framework Core has quite a few differences.

## Need help?

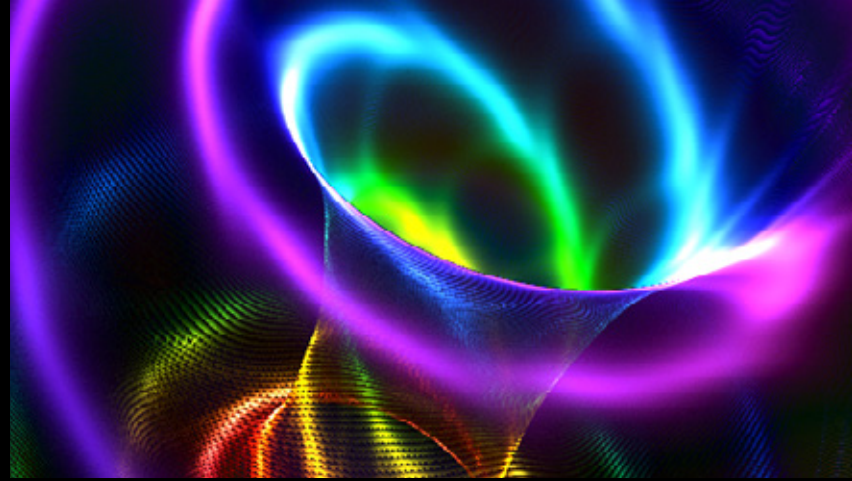
If you are working with .NET Core and have questions, try joining community Slack account:

<http://aspnetcore.slack.com> at <http://tattoocoder.com/aspnet-slack-sign-up/>.



# Build/better

©2017 Stackify



Stackify exists to increase developers' ability to create amazing applications as quickly as possible. We believe that the ideal development team, today and in the future, is consistently optimizing its output across the entire lifecycle of an application; from development to testing to production. Stackify's mission is to give developer teams easy access to powerful tools, which enable them to take the lead in delivering the best applications as quickly as possible.

If you're a developer, team lead, or architect, Stackify's tools were built for you. In fact, have two game-changing, code performance products no developer or dev team should ever be without.



Prefix FREE

Prefix is a popular developer tool for finding and fixing bugs while you write your code. You have profilers and debuggers, but nothing is like Prefix, and it's free!



Retrace FREE TRIAL. STARTS AT \$10/MO

Retrace is an APM tool built specifically for developers and dev teams. Our agent can install on pre-prod or production servers to make black boxes transparent. Try Retrace free for 14-days.

*"We're using Stackify to find potential misbehaviour in our software and to improve the performance of our support team. Most of our incoming tickets can be solved or least linked to our trace/error logs in Retrace.*

*As Stackify supports diagnostic contexts, it is easy for us to filter the logs by LEVEL or customer and to retrieve valuable metadata about the log context.*

*– Julian Neuhaus*

*There is good documentation how to integrate Stackify in your business applications. The 'Error View' is one of our most used features and helps a lot to find, analyze and keep track of bugs.*

*If you're using services/tools like Docker, AWS, Azure or Bluemix it is a must have."*