Developer Network
(https://msdn.microsoft.com/en-us)

Sign in (https://login
wa=wsignin1.0&rpsnv=13&ct=1506473319&rver=6.7.6640.0&wp=MCMBI&wlcxt=MSDN%24MSDN%24MSDN&wreply=https%3a%2f%2fmsdn.mi
us%2fmagazine%2fmt842504%3fMC%3dAppDnDTool%2cASPNET%2cTesting%2cMSAzure%2cDevOps&lc=1033&id=25
Subscriber portal

(https://my.visualstudio.com/?wt.mc_id=o~msft~msdn~nav~subscriber&campaign=o~msft~msdn~nav~subscriber)
Get tools (https://www.visualstudio.com/free-developer-offers/)

magazine (https://msdn.microsoft.com/en-us/magazine/)

Issues and downloads Subscribe Submit article

Issues and downloads / 2017 (https://msdn.microsoft.com/en-us/magazine/mt745087.aspx) / September 2017 (https://msdn.microsoft.com/en-us/magazine/0917mag.aspx) / ASP.NET Core - Getting Started with ASP.NET Core 2.0 (https://msdn.microsoft.com/en-us/magazine/mt842504.aspx)

SEPTEMBER 2017 VOLUME 32 NUMBER 9

ASP.NET Core - Getting Started with ASP.NET Core 2.0

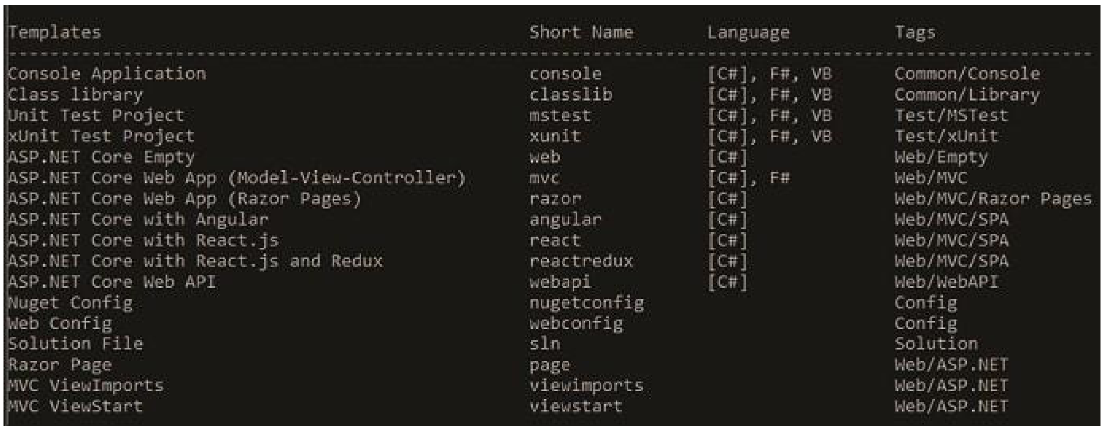
By Mike Rousos (https://msdn.microsoft.com/en-us/magazine/mt149362?author=Mike+Rousos) | September 2017

ASP.NET Core makes it easy to create fast, portable, cross-platform Web applications. This article will guide you through developing a simple ASP.NET Core Web site and will show what role each file in the project plays. It will also explain important ASP.NET Core concepts along the way. There will be a special focus on changes in ASP.NET Core 2.0 to help readers familiar with ASP.NET Core 1.0 and 1.1 make the transition to 2.0.

Creating an ASP.NET Core Project

ASP.NET Core projects can be created from templates using either Visual Studio or the .NET Core command-line interface (.NET CLI). Visual Studio 2017 gives a great .NET Core development experience—with top-notch debugging, Docker integration and many other features—but I'll use the .NET CLI and Visual Studio Code in this walk-through, in case some of you want to follow along on a Mac or Linux dev machine.

The dotnet new command is used to create new .NET Core projects. Running dotnet new without any additional arguments will list the available project templates, as shown in **Figure 1**. If you're familiar with previous versions of the .NET CLI, you'll notice a number of new templates available in version 2.0.



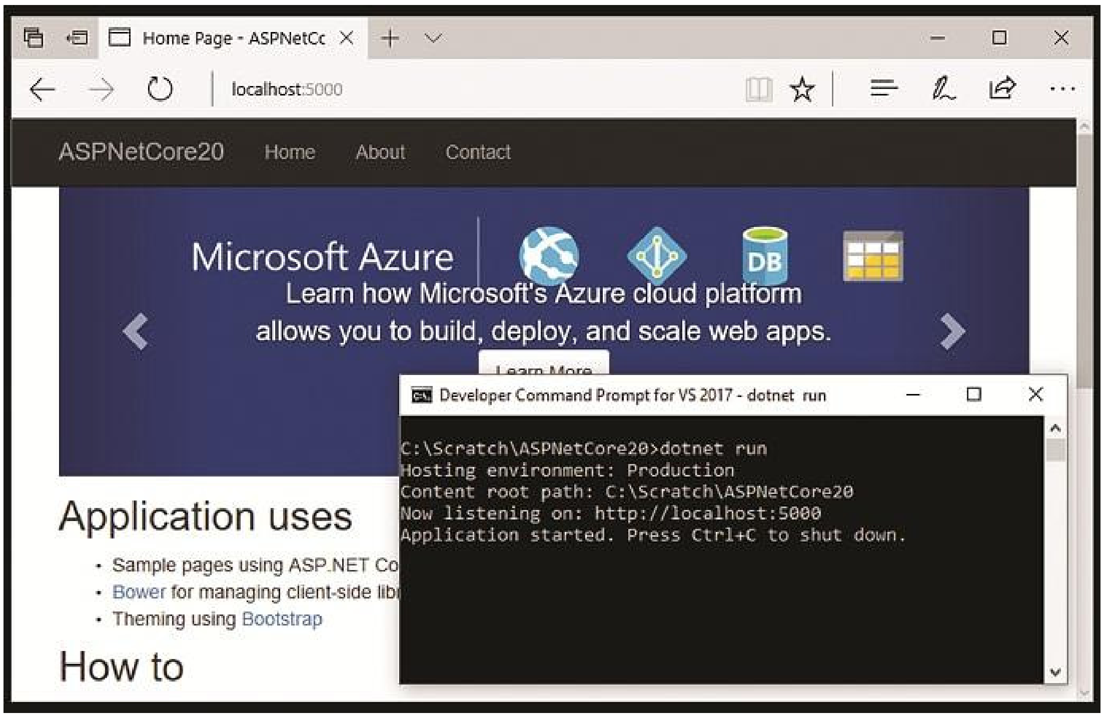
```
Templates                                         Short Name       Language          Tags
-----------------------------------------------------------------------------------------------------------
Console Application                               console          [C#], F#, VB      Common/Console
Class library                                     classlib         [C#], F#, VB      Common/Library
Unit Test Project                                 mstest           [C#], F#, VB      Test/MSTest
xUnit Test Project                                xunit            [C#], F#, VB      Test/xUnit
ASP.NET Core Empty                                web              [C#]              Web/Empty
ASP.NET Core Web App (Model-View-Controller)      mvc              [C#], F#          Web/MVC
ASP.NET Core Web App (Razor Pages)                razor            [C#]              Web/MVC/Razor Pages
ASP.NET Core with Angular                         angular          [C#]              Web/MVC/SPA
ASP.NET Core with React.js                        react            [C#]              Web/MVC/SPA
ASP.NET Core with React.js and Redux              reactredux       [C#]              Web/MVC/SPA
ASP.NET Core Web API                              webapi           [C#]              Web/WebAPI
Nuget Config                                      nugetconfig                        Config
Web Config                                        webconfig                          Config
Solution File                                     sln                                Solution
Razor Page                                        page                               Web/ASP.NET
MVC ViewImports                                   viewimports                        Web/ASP.NET
MVC ViewStart                                     viewstart                          Web/ASP.NET
```

Figure 1 New .NET Core Project Templates

Angular and React.js SPA Templates: These templates create an ASP.NET Core application that serves a single-page application (using either Angular 4 or React.js) as its front end. The templates include both the front-end and back-end applications, as well as a Webpack configuration to build the front end (and csproj modifications to kick off the Webpack build every time the ASP.NET Core project is built).

ASP.NET Core Web App with Razor Pages: Razor Pages is a new ASP.NET Core 2.0 feature that allows you to create pages that can handle requests directly (without needing a controller). These are a great option for scenarios that fit a page-based programming model.

For this walk-through, let's start with the new Razor Pages template by executing dotnet new razor. As soon as the project has been created, you should be able to run it by executing dotnet run. In previous versions of .NET Core, it would've been necessary to execute dotnet restore first to install the necessary NuGet packages. But beginning with .NET Core 2.0, the restore command is now automatically run by CLI commands that depend on it. Go ahead and test the template Web site out by executing dotnet run and navigating to the URL on which the app is listening (likely http://localhost:5000 (http://localhost:5000)). You should see the Web app rendered (as in **Figure 2**).



Learn how Microsoft's Azure cloud platform allows you to build, deploy, and scale web apps.

Application uses

- Sample pages using ASP.NET Co
- Bower for managing client-side libr
- Theming using Bootstrap

How to

```
C:\Scratch\ASPNetCore20>dotnet run
Hosting environment: Production
Content root path: C:\Scratch\ASPNetCore20
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

Figure 2 A Simple ASP.NET Core App Running

Congratulations on launching your first ASP.NET Core 2.0 app! Now that you have a simple Web app running, let's take a look at the project's contents to get a better understanding of how ASP.NET Core works.

Dependencies, Sources and Resources

The first file to look at is the project file itself—the .csproj file. This file tells MSBuild how to build the project—which packages it depends on, what version of .NET Core to target, and so forth. If you've looked at .csproj files in the past, you'll notice this project file is much smaller. A lot of effort has gone into making .csproj files shorter and more human-readable. One notable change that helped with shrinking the project file is that source files no longer need to be listed explicitly. Instead, the .NET Core SDK will automatically compile any .cs files next to the project file or in any directories under the .csproj's directory. Similarly, any .resx files found will be embedded as resources. If you'd prefer to not have all of these .cs files compiled, you can either remove them from the Compile ItemGroup or disable default compile items entirely by setting the EnableDefaultCompileItems property to false.

The .NET version that the app should run against is specified by the <TargetFramework> element. This is set to netcoreapp2.0 in order to take advantage of new ASP.NET Core 2.0 features and the much larger .NET Core 2.0 surface area.

The <PackageReference> element in the middle of the .csproj file indicates a NuGet package on which the project depends. You'll notice in ASP.NET Core 2.0 that you now include just one meta--package by default (Microsoft.AspNetCore.All). This package includes all the other Microsoft.AspNetCore packages in one succinct reference and makes ASP.NET Core 2.0 project files much smaller than project files from previous versions. Additional NuGet dependencies can be added by adding more <PackageReference> elements, by using the Visual Studio NuGet Package management UI, or with the .NET CLI dotnet add command.

If you were using an SPA template (angular, react or reactredux), there would also be custom targets defined in the .csproj file to make sure that Webpack was run when building the project.

Creating and Running the Web Host

Program.cs represents the entry point of the application. ASP.NET Core apps are console applications so, like all console apps, they have a Main method that starts when the app executes.

The contents of the ASP.NET Core 2.0 templates' Main methods are pretty simple—they create an IWebHost object and call Run on it.

If you previously used ASP.NET Core 1.0, you'll notice that this file is a bit simpler than the program.cs from those older templates. The reason for this is the new WebHost.CreateDefaultBuilder method. Previously, the ASP.NET Core app's Main method would configure a WebHostBuilder in order to create the IWebHost instance. This configuration included steps such as specifying a Web server, setting the content root path and enabling IIS integration.

The new CreateDefaultBuilder method simplifies things by creating a ready-to-go IWebHost with most common configuration already done. In addition to specifying the items listed previously, CreateDefaultBuilder also takes care of some setup that was previously handled in Startup.cs (setting up configuration information and registering default logging providers). Because ASP.NET Core is open source, you can see all the details of what CreateDefaultBuilder is doing by viewing its source on GitHub (bit.ly/2uR1Dar (https://bit.ly/2uR1Dar)) if you're interested.

Let's briefly take a look at the most important calls made in CreateDefaultBuilder and their purpose. Although these are all made for you (by CreateDefaultBuilder), it's still good to understand what's happening behind the scenes.

UseKestrel specifies that your application should use Kestrel, a libuv-based cross-platform Web server. The other option here would be to use HttpSys as the Web server (UseHttpSys). HttpSys is supported only on Windows (Windows 7/2008 R2 and later), but has the advantages of allowing Windows authentication and being safe to run directly exposed to the Internet (Kestrel, on the other hand, should sit behind a reverse proxy like IIS, Nginx or Apache if it will receive requests from the Internet).

UseContentRoot specifies the root directory for the application where ASP.NET Core will find site-wide content like config files. Note that this is not the same as the Web root (where static files will be served from), though by default the Web root is based on the content root ([ContentRoot]/wwwroot).

ConfigureAppConfiguration creates the configuration object the app will use to read settings at run time. When called from CreateDefaultBuilder, this will read application configuration settings from an appsettings.json file, an environment-specific .json file (if one exists), environment variables and command-line arguments. If in a development environment, it will also use user secrets. This method is new in ASP.NET Core 2.0 and I'll discuss it in more detail later.

ConfigureLogging sets up logging for the application. When called from CreateDefaultBuilder, console and debug logging providers are added. Like ConfigureAppConfiguration, this method is new and is discussed more later.

UseIISIntegration configures the application to run in IIS. Note that UseKestrel is still needed. IIS is acting as the reverse proxy and Kestrel is still used as the host. Also, UseIISIntegration won't have any effect if the app isn't running behind IIS, so it's safe to call even if the app will be run in non-IIS scenarios.

In many cases, the default configuration provided by CreateDefaultBuilder will be sufficient. All that's needed beyond that call is to specify the Startup class for your application with a call to UseStartup<T>, where T is the Startup type.

If CreateDefaultBuilder doesn't meet your scenario's needs, you should feel free to customize how the IWebHost is being created. If you need to make only small tweaks, you can call CreateDefaultBuilder and then modify the WebHostBuilder that's returned (perhaps by calling ConfigureAppConfiguration again, for example, to add more configuration sources). If you need to make larger changes to the IWebHost, you can skip calling CreateDefaultBuilder completely and just construct a WebHostBuilder yourself, as you would have with ASP.NET Core 1.0 or 1.1. Even if you go this route, you can still take advantage of the new ConfigureAppConfiguration and ConfigureLogging methods. More details on Web host configuration are available at bit.ly/2uuSwwM (https://bit.ly/2uuSwwM).

ASP.NET Core Environments

A couple of actions CreateDefaultBuilder takes depend on in which environment your ASP.NET Core application is running. The concept of environments isn't new in 2.0, but is worth briefly reviewing because it comes up frequently.

In ASP.NET Core, the environment an application is running in is indicated by the ASPNETCORE_ENVIRONMENT environment variable. You can set this to any value you like, but the values Development, Staging and Production are typically used. So, if you set the ASPNETCORE_ENVIRONMENT variable to Development prior to calling dotnet run (or if you set that environment variable in a launchSettings.json file), your app will run in Development mode (instead of Production, which is the default without any variables set). This value is used by several ASP.NET Core features (I'll reference it when discussing Configuration and Logging, later) to modify runtime behavior and can be accessed in your own code using the IHostingEnvironment service. More information on ASP.NET Core environments is available in the ASP.NET Core docs (bit.ly/2eICDMF (https://bit.ly/2eICDMF)).

ASP.NET Core Configuration

ASP.NET Core uses the Microsoft.Extensions.Configuration package's IConfiguration interface to provide runtime configuration settings. As mentioned previously, CreateDefaultBuilder will read settings from .json files and environment variables. The configuration system is extensible, though, and can read configuration information from a wide variety of providers (.json files, .xml files, .ini files, environment variables, Azure Key Vault and so forth.).

When working with IConfiguration and IConfigurationBuilder objects, remember that the order the providers are added is important. Later providers can override settings from previous providers, so you'll want to add common base providers first and then, later, add environment-specific providers that might override some of the settings.

Configuration settings in ASP.NET Core are hierarchical. In the new project you created, for example, appsettings.json (see **Figure 3**) contains a top-level Logging element with sub-settings underneath it. These settings indicate the minimum priority of messages to log (via the "LogLevel" settings) and whether the app's logical scope at the time the message is logged should be recorded (via IncludeScopes). To retrieve nested settings like these, you can either use the IConfiguration.GetSection method to retrieve a single section of the configuration or specify the full path of a particular setting, delimited with colons. So, the value of IncludeScopes in the project could be retrieved as:

Figure 3 ASP.NET Core Settings File

```
{
  "Logging": {
    "IncludeScopes": false,
    "Debug": {
      "LogLevel": {
        "Default": "Warning"
      }
    },
    "Console": {
      "LogLevel": {
        "Default": "Warning"
      }
    }
  }
}
```

```
Configuration["Logging:IncludeScopes"]
```

When defining configuration settings with environment variables, the environment variable name should include all levels of the hierarchy and can be delimited with either a colon (:) or double underscore (__). For example, an environment variable called Logging__IncludeScopes would override the IncludeScopes setting of the example file in **Figure 3**, assuming that the environment variable provider is added after the settings file, like it is in the CreateDefaultBuilder case.

Because WebHost.CreateDefaultBuilder is reading configuration from both appsettings.json and environment-specific .json files, you'll notice that logging behavior changes when you change environments (appsettings.Development.json overrides the default appsettings.json's LogLevel settings with more verbose "debug" and "information" levels). If you set the environment to Development prior to calling dotnet run, you should notice a fair bit of logging happening to the console (which is great because it's useful for debugging). On the other hand, if you set the environment to Production, you won't get any console logging except for warnings and errors (which is also great because console logging is slow and should be kept to a minimum in production).

If you have experience with ASP.NET Core 1.0 and 1.1, you might notice that the ConfigureAppConfiguration method is new to 2.0. Previously, it was common to create an IConfiguration as part of the Startup type's creation. Using the new ConfigureAppConfiguration method is a useful alternative because it updates the IConfiguration object stored in the application's dependency injection (DI) container for easy future retrieval and makes configuration settings available even earlier in your application's lifetime.

ASP.NET Core Logging

As with configuration setup, if you're familiar with earlier versions of ASP.NET Core you might remember logging setup being done in Startup.cs instead of in Program.cs. In ASP.NET Core 2.0, logging setup can now be done when building an IWebHost via the ConfigureLogging method.

It's still possible to set up logging in Startup (using services.Add-Logging in Startup.ConfigureServices), but by configuring logging at Web host-creation time, the Startup type is streamlined and logging is available even earlier in the app startup process.

Also like configuration, ASP.NET Core logging is extensible. Different providers are registered to log to different endpoints. Many providers are available immediately with a reference to Microsoft.AspNetCore.All, and even more are available from the .NET developer community.

As can be seen in WebHost.CreateDefaultBuilder's source code, logging providers can be added by calling provider-specific extension methods like AddDebug or AddConsole on ILoggingBuilder. If you use WebHost.CreateDefaultBuilder but still want to register logging providers other than the default Debug and Console ones, it's possible to do so with an additional call to ConfigureLogging on the IWebHostBuilder returned by CreateDefaultBuilder

Once logging has been configured and providers registered, ASP.NET Core will automatically log messages regarding its work to process incoming requests. You can also log your own diagnostic messages by requesting an ILogger object via dependency injection (more on this in the next section). Calls to ILogger.Log and level-specific variants (like LogCritical, LogInformation and so on) are used to log messages.

The Startup Type

Now that you've looked at Program.cs to see how the Web host is created, let's jump over to Startup.cs. The Startup type that your app should use is indicated by the call to UseStartup when creating the IWebHost. Not a lot has changed in Startup.cs in ASP.NET Core 2.0 (except for it becoming simpler because logging and configuration are set up in Program.cs), but I'll briefly review the Startup type's two important methods because they're so central to an ASP.NET Core app.

Dependency Injection The Startup type's ConfigureServices method adds services to the application's dependency injection container. All ASP.NET Core apps have a default dependency injection container to store services for later use. This allows services to be made available without tight coupling to the components that depend on them. You've already seen a couple examples of this—both ConfigureAppConfiguration and ConfigureLogging will add services to the container for use later in your application. At run time, if an instance of a type is called for, ASP.NET Core will automatically retrieve the object from the dependency injection container if possible.

For example, your ASP.NET Core 2.0 project's Startup class has a constructor that takes an IConfiguration parameter. This constructor will be called automatically when your IWebHost begins to run. When that happens, ASP.NET Core will supply the required IConfiguration argument from the dependency injection container.

As another example, if you want to log messages from a Razor Page, you can request a logger object as a parameter to the page model's constructor (like how Startup requests an IConfiguration object) or in cshtml with the @inject syntax, as the following shows:

```
@using Microsoft.Extensions.Logging
@inject ILogger<Index_Page> logger

@functions {
  public void OnGet()
    {
      logger.LogInformation("Beginning GET");
    }
}
```

Something similar could be done to retrieve an IConfiguration object or any other type that has been registered as a service. In this way, the Startup type, Razor Pages, controllers and so forth, can loosely depend on services provided in the dependency injection container.

As mentioned at the beginning of this section, services are added to the dependency injection container in the Startup.ConfigureServices method. The project template you used to create your application already has one call in ConfigureServices: services.AddMvc. As you might guess, this registers services needed by the MVC framework.

Another type of service that's common to see registered in the ConfigureServices method is Entity Framework Core. Although it's not used in this sample, apps that make use of Entity Framework Core typically register the DbContexts needed for working with Entity Framework models using calls to services.AddDbContext.

You can also register your own types and services here by calling services.AddTransient, services.AddScoped or services.AddSingleton (depending on the lifetime required for dependency injection-provided objects). Registering as a singleton will result in a single instance of the service that's returned every time its type is requested, whereas registering as transient will cause a new instance to be created for each request. Adding as scoped will cause a single instance of a service to be used throughout the processing of a single HTTP request. More details on dependency injection in ASP.NET Core are available at bit.ly/2w7XtJI (https://bit.ly/2w7XtJI).

HTTP Request-Processing Pipeline and Middleware The other important method in the Startup type is the Configure method. This is where the heart of the ASP.NET Core application—its HTTP request-processing pipeline—is set up. In this method, different pieces of middleware are registered that will act on incoming HTTP requests to generate responses.

In Startup.Configure, middleware components are added to an IApplicationBuilder to form the processing pipeline. When a request comes in, the first piece of middleware registered will be invoked. That middleware will perform whatever logic it needs to and then either call the next piece of middleware in the pipeline or, if it has completely handled the response, return to the previous piece of middleware (if there was a previous one) so that it can execute any logic necessary after a response has been prepared. This pattern of calling middleware components in order when a request arrives and then in reverse order after it has been handled is illustrated in **Figure 4**.

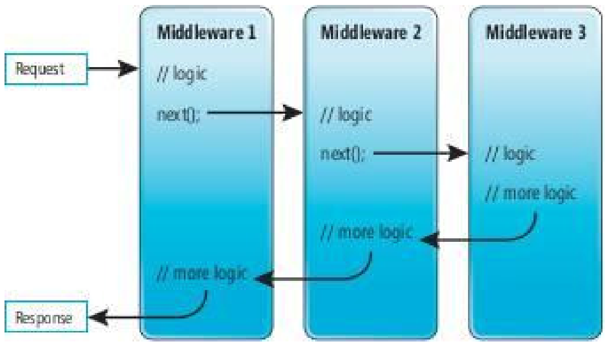


Figure 4 ASP.NET Core Middleware Processing Pipeline

To take a concrete example, **Figure 5** shows the Configure method from the template project. When a new request comes in, it will go first to either the DeveloperExceptionPage middleware or the ExceptionHandler middleware, depending on your environment (as before, this is configured with the ASPNETCORE_ENVIRONMENT environment variable). These middleware components won’t take much action initially, but after subsequent middleware has run and the request is on its way back out of the middleware pipeline, they will watch for and handle exceptions.

Figure 5 ASP.NET Core Startup.Configure Method Sets Up the Middleware Pipeline

```
public void Configure(IApplicationBuilder app, IHostingEnvironmen
{
  if (env.IsDevelopment())
  {
    app.UseDeveloperExceptionPage();
  }
  else
  {
    app.UseExceptionHandler("/Error");
  }

  app.UseStaticFiles();

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

Next, StaticFiles middleware will be called, which may serve the request by providing a static file (an image or style sheet, for example). If it does, it will halt the pipeline and return control to the previous middleware (the exception handlers). If the StaticFiles middleware can't provide a response, it will call the next piece of middleware—the MVC middleware. This middleware will attempt to route the request to an MVC controller (or Razor Page) for fulfillment, according to the routing options specified.

The order the middleware components are registered is very important. If UseStaticFiles came after UseMvc, the application would try to route all requests to MVC controllers before checking for static files. That could result in a noticeable perf degrade! If the exception-handling middleware came later in the pipeline, it wouldn't be able to handle exceptions occurring in prior middleware components.

Razor Pages

Besides the .csproj, program.cs, and startup.cs files, your ASP.NET Core project also contains a Pages folder containing the application's Razor Pages. Pages are similar to MVC views, but requests can be routed directly to a Razor Page without the need for a separate controller. This makes it possible to simplify page-based applications and keep views and view models together. The model supporting the page can be included in the cshtml page directly (in a @functions directive), or in a separate code file, which is referenced with the @model directive.

To learn more about Razor Pages, check out Steve Smith's article, "Simpler ASP.NET MVC Apps with Razor Pages," also in this issue.

Wrapping Up

Hopefully this walk-through helps explain how to create a new ASP.NET Core 2.0 Web application and demystifies the contents of the new project templates. I've reviewed the project contents from the streamlined .csproj file, to the application entry point and Web host configuration in Program.cs, to service and middleware registration in Startup.cs.

To keep digging deeper into what's possible with ASP.NET Core, it might be useful to create some new projects using some of the other templates, like the Web API template or perhaps some of the new SPA templates. You may also want to try out deploying your ASP.NET Core app to Azure as an App Service Web App or packaging the application as a Linux or Windows Docker image. And, of course, please check out the full documentation at docs.microsoft.com/aspnet/core (https://msdn.microsoft.com/en-us/magazine/core) for more information on the topics covered in this article and more.

Mike Rousos is a principal software engineer on the .NET Customer Success Team. Rousos has been a member of the .NET team since 2004, working on technologies including tracing, managed security, hosting and, most recently, .NET Core.

Thanks to the following Microsoft technical experts who reviewed this article: Glenn Condron and Ryan Nowak

---