Porting existing ASP.NET apps to .NET Core

Steve "ardalis" Smith

Authors:

Steve "ardalis" Smith, Software Architect and Trainer - Ardalis.com

Participants and Reviewers:

Nish Anil, Senior Program Manager, .NET team, Microsoft

Mike Rousos, Principal Software Engineer, .NET team, Microsoft

Scott Addie, Senior Content Developer, .NET team, Microsoft

David Pine, Senior Content Developer, .NET team, Microsoft

Version

This guide covers **.NET Core 3.1** and updates related to the same technology "wave" (that is, Azure and other third-party technologies) coinciding in time with the .NET Core 3.1 release. Updating from .NET Core 3.1 to .NET 5.0 (the next version) is relatively straightforward and certainly will require substantially less effort than porting from .NET Framework to .NET Core. Migrating from .NET Framework 4.x to .NET 5.0 will be similar to migrating to .NET Core 3.1. For more information, see choosing the right .NET Core version.

Who should use this guide

This guide's audience is developers, development leads, and architects who are interested in migrating their existing apps written for ASP.NET MVC and Web API (.NET Framework 4.x) to .NET Core. ASP.NET Web Forms developers will benefit from this guide but should also read the Blazor for ASP.NET Web Forms Developers e-book.

A secondary audience is technical decision-makers planning when to move their apps to .NET Core.

The target audience for this book is .NET developers with large, existing apps that run on ASP.NET MVC and Web API. Apps built on ASP.NET Web Forms are outside of the focus of this book, though much of the information comparing .NET Framework and .NET Core may still be relevant.

How you can use this guide

You can read this book straight through, as we expect many readers to do. This book will provide you first with considerations for whether you should port your app at all. That content is followed by architectural differences between .NET Framework and .NET Core. From there, you'll learn strategies for migrating a large solution over time and how to port a real app. Next, the book includes deployment scenarios that address the need to run different apps while appearing as a single app to users. The book concludes with two case studies describing real apps that have migrated from ASP.NET MVC to ASP.NET Core.

Whether or not you choose to start from the first chapter, you can reference any of these chapters to learn about specific concepts:

- Architectural differences
- Migrate large solutions
- Sample migration
- Deployment scenarios

This guide is available both in PDF form and online. Feel free to forward this document or links to its online version to your team to ensure a common understanding of these concepts.

Send your feedback

This book and related samples are constantly evolving, so your feedback is welcomed! If you have comments about how this book can be improved, use the feedback section at the bottom of any page built on GitHub issues.

Contents

Summary 72

Introduction to porting apps to .NET Core

.NET Core is a revolutionary step forward from .NET Framework. It offers a host of advantages over .NET Framework across the board from productivity to performance, from cross-platform support to developer satisfaction. ASP.NET Core was even voted the most-loved web framework in the 2020 Stack Overflow developer survey. Clearly there are strong reasons to consider migrating.

It's important to keep in mind that .NET Core is the Future of .NET. To quote this article:

New apps should be built on .NET Core. .NET Core is where future investments in .NET will happen. Existing apps are safe to remain on .NET Framework which will be supported. Existing apps that want to take advantage of the new features in .NET should consider moving to .NET Core. As we plan into the future, we will be bringing in even more capabilities to the platform.

However, upgrading your app to ASP.NET Core will require some effort. That effort should be balanced against business value and goals. .NET Framework apps have a long life ahead of them, with support built into Windows for the foreseeable future. What are some of the questions you should consider before deciding migration to .NET Core is appropriate? What are the expected advantages? What are the tradeoffs? How much effort is involved? These obvious questions are just the beginning, but make for a great starting point as teams consider how to support their customers' needs with apps today and tomorrow.

- Is migration to .NET Core appropriate?
- When does it make sense to remain on .NET Framework?
- Should apps target ASP.NET Core 2.1 as a stepping stone?
- How should teams choose the right .NET Core version to target?
- What strategies are recommended for incremental migration of large apps?
- What deployment strategies should be considered when porting to .NET Core?
- Where can we find additional resources?

This introductory chapter addresses all of these questions and more before moving on to more specific and technical considerations in future chapters.

References

- 2020 Stack Overflow developer survey most loved web frameworks

- [.NET Core is the Future of .NET](#)

Migration considerations

The most fundamental question teams must answer when it comes to porting their apps to .NET Core is, should they port at all? In some cases, the best path forward is to remain on .NET Framework using ASP.NET MVC and/or Web API. This chapter considers reasons why moving to .NET Core makes sense. The chapter also considers scenarios and counterpoints for staying on .NET Framework.

Is migration to .NET Core appropriate?

Let's start with some of the reasons why you might want to move to .NET Core. There are quite a few, so don't consider this list exhaustive.

Cross-platform support

Apps built on .NET Core are truly cross-platform and can run on Windows, Linux, and macOS. Not only can your developers use whatever hardware they want, but you can also host your app anywhere. Examples range from local IIS to Azure in the cloud or from Linux Docker containers to IoT devices.

Performance and scalability

Apps built with .NET Core are running on [one of the fastest tech stacks available anywhere](#). ASP.NET MVC apps often see performance improvements on ASP.NET Core, especially if they're updated to take advantage of some new features available in .NET Core.

Cloud-native

For the above reasons and others, .NET Core apps are well-suited to running in cloud hosting environments. Lightweight and fast, .NET Core apps can be deployed to Azure App Services or containers and scaled horizontally as needed to meet immediate system demand.

Maintainable

For many apps, while they've continued to meet customer and business needs, technical debt has accumulated and maintaining the app has grown expensive. ASP.NET Core apps are more easily tested than ASP.NET MVC apps, making them easier to refactor and extend with confidence.

Modular

ASP.NET Core is modular, using NuGet packages as a first-class part of the framework. Apps built for .NET Core all support dependency injection, making it easy to compose solutions from whatever implementations are needed for a given environment. Building microservices with .NET Core is easier than with ASP.NET MVC with its dependency on IIS, which opens up additional options to break up large apps into smaller modules.

Modern

Staying on a modern, actively developed technology stack has a host of advantages. New features and C# language features will only be added to .NET Core. The .NET Framework has had its last release with version 4.8, and versions of C# beyond 8 won't target .NET Framework. While ASP.NET MVC will remain supported by Microsoft for many years, the best and brightest .NET software developers are likely looking to use the more modern .NET Core framework, with all of the advantages it offers (only some of which are summarized above). Finding developers with the skills to maintain an ASP.NET MVC app will start to become a challenge at some point, as will finding online training and troubleshooting assistance. There probably aren't that many new blog posts being written about ASP.NET MVC 5, while there are plenty being written for .NET 5.0, for example.

There are many compelling reasons to consider migrating to .NET Core, which presumably is why you're reading this book! But let's consider some disadvantages and reasons why it may make more sense to remain on the .NET Framework.

When is .NET Framework appropriate?

The biggest reason to stay on .NET Framework is when an app isn't under active development and wouldn't benefit substantially from the advantages listed above. In that case, there probably isn't a good business case to incur the cost of porting the app. If your app might benefit from the advantages .NET Core offers, you may still need to stay on .NET Framework if you need certain technologies that are unavailable on .NET Core. There are some .NET technologies that are unavailable on .NET Core, including AppDomains, Remoting, Code Access Security (CAS), Security Transparency, and System.EnterpriseServices. A brief summary of these technologies and their alternatives is included here. For more detailed guidance, see the documentation.

Application domains

Application domains (AppDomains) isolate apps from one another. AppDomains require runtime support and can be expensive. Creating additional app domains isn't supported, and there are no plans to add this capability to .NET Core in the future. For code isolation, use separate processes or containers as an alternative.

WCF

Server-side WCF isn't supported in .NET Core. .NET Core supports WCF clients but not WCF hosts. Apps that require this functionality will need to upgrade to a different communication technology (such as gRPC or REST) as part of a migration.

There is a WCF client port available from the .NET Foundation. It is entirely open source, cross platform, and supported by Microsoft. There is also a community-supported CoreWCF project that is *not* officially supported by Microsoft.

Remoting

.NET Remoting was identified as a problematic architecture. It's used for cross-AppDomain communication, which is no longer supported. Also, Remoting requires runtime support, which is

expensive to maintain. For these reasons, .NET Remoting isn't supported on .NET Core, and the product team doesn't plan on adding support for it in the future. There are several alternative messaging strategies and implementations you can use to replace remoting in your .NET Core apps.

Code Access Security (CAS) and Security Transparency

Neither of these technologies are supported by .NET Core. Instead, the recommendation is to use security boundaries provided by the operating system. For example, virtualization, containers, or user accounts. Run processes with the minimal set of privileges necessary.

References

[.NET Framework Technologies Unavailable on .NET Core](#)

Migrate to ASP.NET Core 2.1

ASP.NET Core 2.1 is an interesting release because it's the only currently supported .NET Core release to support both .NET Core and .NET Framework runtimes. As such, it may offer an easier upgrade path for some apps when compared to upgrading all parts of the app to .NET Core at once. As an LTS release, support for .NET Core 2.1 will continue through August 2021. Support for ASP.NET Core 2.1 running on .NET Framework will continue for as long as its underlying .NET Framework is supported.

Should apps run on .NET Framework with ASP.NET Core 2.1?

ASP.NET Core 2.2 and earlier supported both .NET Core and .NET Framework runtimes. Does it make sense to migrate some or all of an app to ASP.NET Core 2.1 as a stepping stone, before porting over completely to .NET Core? Apps, or subsets of apps, could see their front-end ASP.NET logic ported to use ASP.NET Core, while still consuming .NET Framework libraries for business logic and infrastructure consumption. This approach may make sense when there's a relatively thin UI layer without much business logic, and a much larger set of functionality in class libraries.

The main benefit of porting just the front-end web layer to ASP.NET Core 2.1 is that the existing .NET class libraries can remain as is during the initial migration. They may be in continued use by other .NET apps or simply don't need to be in scope for the first iteration of a planned full migration to .NET Core. Reducing the scope of the initial migration for large apps helps provide incremental goals that act as stepping stones toward the desired end state, which is often a complete port to .NET Core.

If you have an existing app that may use this strategy, some things you can do today to help prepare for the process are to move as much business logic, data access, and other non-UI logic out of the ASP.NET projects and into separate class libraries as possible. It will also help if you have automated test coverage of your system, so that you can verify behavior remains consistent before and after the migration.

If your app is so large that you can't migrate the entire web app at once, and you need to be able to deploy the new ASP.NET Core app side-by-side with the existing ASP.NET app, there are deployment strategies that can be used to achieve this. These are covered in [Chapter 5: Deployment Scenarios](#).

Keep in mind that ASP.NET Core 2.1 is the last LTS release of .NET Core that supports running on .NET Framework and consuming .NET Framework libraries. This release will soon be unsupported, but ASP.NET Core 2.1 on .NET Framework will be supported as long as the .NET Framework (even after .NET Core 2.1 support ends). For more information, see [ASP.NET Core 2.1 on .NET Framework](#).

References

[Migrating from ASP.NET to ASP.NET Core 2.1](#)

Choose the right .NET Core version

The largest consideration for most organizations when choosing a version of .NET Core to target is the support lifecycle. Long Term Support (LTS) releases ship less frequently but have a longer support window than Current (non-LTS) releases. Currently, LTS releases are scheduled to ship every other year. Customers can choose which releases to target, and can install different releases of .NET Core side by side on the same machine. LTS releases will receive only critical and compatible fixes throughout their lifecycle. Current releases will receive these same fixes and will also be updated with compatible innovations and features. LTS releases are supported for three years after their initial release. Current releases are supported for three months after a subsequent Current or LTS release.

Most customers looking to migrate a large .NET Framework app to .NET Core today are probably looking for a stable destination, given that they haven't already made the move to an earlier version of .NET Core. In this case, the best .NET Core version to target for the migration is .NET Core 3.1, which is the current LTS version. Support for .NET Core 3.1 ends in December 2022. The next planned LTS release will be .NET 6.0, slated to ship in November 2021.

Updating from .NET Core 3.1 to .NET 5.0 (the next version) requires much less effort than porting from .NET Framework to .NET Core. For this reason, the recommendation for most customers is to upgrade to .NET Core 3.1 first. Then decide whether the next update should be to the latest current release (.NET 5.0) or to wait for the next LTS release (.NET 6.0) before upgrading further.

This book assumes .NET Framework apps will be upgraded to .NET Core 3.1.

References

[.NET Core and .NET 5 Support Policy](#)

Strategies for migrating incrementally

The biggest challenge with migrating any large app is determining how to break the process into smaller tasks. There are several strategies that can be applied for this purpose, including breaking the app into horizontal layers such as UI, data access, business logic, or breaking up the app into separate, smaller apps. Another strategy is to upgrade some or all of the app to different framework versions on the way to a recent .NET Core release.

Consider the challenge of migrating a large ASP.NET 4.5 app. One approach is to migrate directly from .NET Framework 4.5 to .NET Core 3.1. However, this approach needs to account for every breaking change between the two frameworks and versions, which are substantial.

One recent addition to the .NET ecosystem that helps with interoperability between different .NET frameworks is .NET Standard. .NET Standard allows libraries to build against the agreed upon set of common APIs, ensuring they can be used in any .NET app. .NET Standard 2.0 is notable because it covers most base class library functionality used by most .NET Framework and .NET Core apps. Unfortunately, the earliest version of .NET with support for .NET Standard 2.0 is .NET Framework 4.6.1, and there are a number of updates in .NET Framework 4.8 that make it a compelling choice for initial upgrades.

One approach to incrementally upgrade a .NET Framework 4.5 system is to first update its class libraries to .NET Framework 4.8. Then, modify these libraries to be .NET Standard class libraries. Use multi-targeting and conditional compilation, if necessary. This step can be helpful in scenarios where app dependencies require .NET Framework and cannot easily be ported directly to use .NET Standard and .NET Core. Since .NET Framework libraries can be consumed by ASP.NET Core 2.1 apps, the next step is to migrate some or all of the web functionality of the app to ASP.NET Core 2.1 (as described in the previous chapter).

Once the app is running on ASP.NET Core 2.1, migrating it to ASP.NET Core 3.1 in isolation is relatively straightforward. The most likely challenge during this step is updating incompatible dependencies to support .NET Core and possibly higher versions of .NET Standard. For apps that don't have problematic dependencies on .NET Framework-only libraries, there's little reason to upgrade to ASP.NET Core 2.1. Porting directly to ASP.NET Core 3.1 makes more sense and requires less effort.

By the time the app is running on .NET Core 3.1, migrating to the current .NET 5.0 release is relatively painless. The process primarily involves updating the target framework of your project files and their associated NuGet package dependencies. While there are several breaking changes to consider, most apps don't require significant modifications to move from .NET Core 3.1 to .NET 5.0. The primary deciding factor in choosing between .NET Core 3.1 and .NET 5.0 is likely to be support.

References

- What is .NET Standard?
- Introducing .NET 5
- Migrate from ASP.NET Core 3.1 to 5.0

Strategies for migrating ASP.NET Web Forms apps

This book offers guidance for migrating large ASP.NET MVC and Web API apps to .NET Core. Some of these ASP.NET apps may also include Web Forms (*.aspx*) pages that must be addressed. ASP.NET Web Forms isn't supported in .NET Core (nor are ASP.NET Web Pages). Typically, the functionality of these pages must be rewritten when porting to ASP.NET Core. There are, however, some strategies you can apply before or during such migration to help reduce the overall effort required.

Web Forms will continue to be supported for quite some time. One option may be to keep this functionality in an ASP.NET 4.x app.

Separate business logic and other concerns

The less code in your ASP.NET Web Forms pages, the better. When possible, keep business logic and other concerns like data access in separate classes, ideally in separate class libraries. These class libraries can be ported to .NET Standard and consumed by any ASP.NET Core app.

Implement client behavior and web APIs

Consider the choice between implementing logic in Web Forms or in the browser with the help of API calls. Favor the latter. Migrating APIs to ASP.NET Core is supported. Client-side behavior should be independent of the server-side stack your app is using. Using this approach has the added benefit of providing a more responsive user experience.

Consider Blazor

Blazor lets you build interactive web UIs with C# instead of JavaScript. It can run on the server or in the browser using WebAssembly. ASP.NET Web Forms apps may be ported page-by-page to Blazor apps. For more information on porting Web Forms apps to Blazor, see Blazor for ASP.NET Web Forms Developers. In addition, many Web Forms controls have been ported to Blazor as part of an open-source community project, Blazor Web Forms Components. With these components, you can more easily port Web Forms pages to Blazor even if they use the built-in Web Forms controls.

Summary

Migrating directly from ASP.NET Web Forms to ASP.NET Core isn't supported. However, there are strategies to make moving to ASP.NET Core easier. You can migrate your Web Forms functionality to ASP.NET Core successfully by:

- Keeping non-web functionality out of your projects.
- Using web APIs wherever possible.
- Considering Blazor as an option for a more modern UI.

References

- Free e-book: Blazor for ASP.NET Web Forms Developers
- Blazor Web Forms Components (Community Project)

Deployment strategies

One consideration as you plan the migration of your large ASP.NET app to ASP.NET Core is how you'll deploy the new app. With ASP.NET, deployment options were limited to IIS on Windows. With ASP.NET Core, a much wider array of deployment options is available.

Cross-platform options

Because .NET Core runs on Linux, you'll find some hosting options available that weren't a consideration previously. Linux-based hosting may be preferable for the following reasons:

- Your organization has infrastructure or expertise.
- Hosting providers offer attractive features or pricing for Linux-based hosting.

.NET Core opens to door to these options.

Cloud native development

Most organizations today are using cloud platforms for at least some of their software capabilities. With an app migration to .NET Core, it's a good time to consider whether the app should be purposefully written with cloud hosting in mind. Such *cloud native* apps are better able to apply cloud capabilities like serverless, microservices, and can be less concerned with the low-level details of how and where they may be deployed.

Learn more about cloud native app development in this free e-book, Architecting Cloud Native .NET Applications for Azure.

Leverage containers

Modern apps often leverage containers as a means of reducing variation between hosting environments. By deploying an app to a particular container, the container-hosted app will run the same whether it's running on a developer's laptop or in production. As part of a migration to .NET Core, it may make sense to consider whether the app would benefit from deployment via container, either as a full monolith or broken up into multiple smaller containerized apps.

Side-by-side deployment options

Migrating large .NET Framework apps to .NET Core often requires a substantial effort. Most organizations will want to be able to break this effort up in some fashion, so that pieces of the app can be migrated and deployed in production before the entire migration is complete. Running both the original ASP.NET app and its newly-migrated ASP.NET Core sub-app(s) side by side is a frequently cited goal. This can be achieved through a number of mechanisms including leveraging IIS routing, which is covered in chapter 5. Other options include leveraging app gateways or cloud design patterns like backends for frontends to manage sets of ASP.NET Web APIs and ASP.NET Core API endpoints.

IIS on Windows

You can continue hosting your apps on IIS running on Windows. This is a fine option for customers who want to take advantage of ASP.NET Core features without changing their current deployment model. While moving to ASP.NET Core provides more options in terms of how and where to deploy your apps, it doesn't require that you change from the status quo of using the proven combination of IIS on Windows.

Other options on Windows

You can host apps side-by-side apps on Windows using any combination of Kestrel, HTTP.sys, and IIS hosts, in addition to Docker containers, if needed. If your app requires a combination of Windows and Linux services, hosting on a Windows server with WSL and/or Linux Docker containers provides a single server solution to hosting all parts of the app.

References

- Host and deploy ASP.NET Core
- Host ASP.NET Core on Windows with IIS
- Troubleshooting ASP.NET Core on Azure App Service and IIS

Additional migration resources

As you're planning and executing your migration from ASP.NET MVC and/or Web API to ASP.NET Core, there are a number of resources available to help beyond this book. Make a note of these and leverage them where appropriate to help you overcome obstacles you encounter on your migration journey.

Official documentation

The official documentation website, docs.microsoft.com, has the most up-to-date information available about versions, frameworks, breaking changes, and support options. You'll find many links in this book to docs articles, but for any problem you're facing it's often worth at least doing a quick search of the docs to see if there is already information covering the issue and offering a solution or workaround.

GitHub

Because .NET Core is an open-source project, many issues are discovered, reported, discussed, and fixed on GitHub. Microsoft has several GitHub organizations in which you'll find repositories that may be helpful. A partial list of these organizations and some of their public repositories are listed below:

- Microsoft
    - ASP.NET API Versioning
- dotnet
    - ASP.NET Core
    - .NET Runtime
    - Entity Framework Core
    - C# Language
    - Docs
    - Docs Samples
    - Try Convert
- .NET Architecture Reference Apps

- eShopModernizing
- eShopOnWeb
- eShopOnContainers

If you run into problems with your migration, these GitHub repositories are a good place to report them. The product teams watch the issues and typically respond quickly to bug reports (though "how to" questions may be more appropriately directed to Stack Overflow).

Stack Overflow

Stack Overflow has a wealth of information in the form of previous questions asked and answers given, with the most helpful answers listed first and marked if they solved the problem. In addition to searching for an existing solution to a problem you may encounter, you can of course also ask a question yourself and hope for some response from the .NET community. Don't forget you can narrow down a search by using tags, and remember to use appropriate tags when you ask questions to maximize the chances of someone with the experience needed noticing your question.

YouTube channels

YouTube has a huge amount of .NET and .NET Core video content, which may include useful tutorials or walkthroughs covering any scenario you may encounter. Consider searching it separately if your other efforts to find help online come up short. Here are a few good places to get started:

- dotnet
- Visual Studio

Twitter, Gitter, Slack, and other community channels

You'll find many other ways to connect with .NET developers on the .NET Community page. You can also join the DotNetEvolution Discord server. Additionally, many product teams and team members are on Twitter as well as in various other communities. You can follow and communicate with the author of this book on Twitter as well.

References

- Overview of porting from .NET Framework to .NET Core
- Migrate from ASP.NET to ASP.NET Core
- .NET Community Resources

Architectural differences between ASP.NET MVC and ASP.NET Core

There are many architectural differences between ASP.NET MVC on .NET Framework and ASP.NET Core. It's important to have a broad understanding of these differences as teams evaluate the work involved in porting their ASP.NET MVC apps to ASP.NET Core. This chapter looks at each of the ways in which ASP.NET Core differs substantially from ASP.NET MVC.

Breaking changes

.NET Core is a cross-platform rewrite of .NET Framework. There are many breaking changes between the two frameworks. The following sections identify specific differences between how ASP.NET MVC and ASP.NET Core apps are designed and developed. Take care to also examine the documentation to determine which framework libraries you're using that may need to change. In many cases, a replacement NuGet package exists to fill in any gaps left between .NET Framework and .NET Core. In rare cases, you may need to find a third-party solution or implement new custom code to address incompatibilities.

App startup differences between ASP.NET MVC and ASP.NET Core

ASP.NET MVC apps lived entirely within Internet Information Server (IIS), the primary web server available on Windows operating systems. Unlike ASP.NET MVC, ASP.NET Core apps are executable apps. You can run them from the command line, using dotnet run. They have an entry point method like all C# programs, typically public static void Main() or a similar variation (perhaps with arguments or async support). This is perhaps the biggest architectural difference between ASP.NET Core and ASP.NET MVC, and is one of several differences that allows ASP.NET Core to run on non-Windows systems.

ASP.NET MVC Startup

Hosted within IIS, ASP.NET apps rely on IIS to instantiate certain objects and call certain methods when a request arrives. ASP.NET creates an instance of the *Global.asax* file's class, which derives from HttpApplication. When the first request is received, before handling the request itself, ASP.NET calls the Application_Start method in the *Global.asax* file's class. Any logic that needs to run when the ASP.NET MVC app begins can be added to this method.

Many NuGet packages for ASP.NET MVC and Web API use the WebActivator package to let them run some code during app startup. By convention, this code would typically be added to an *App_Start* folder and would be configured via attribute to run either immediately before or just after Application_Start.

It's also possible to use the Open Web Interface for .NET (OWIN) and Project Katana with ASP.NET MVC. When doing so, the app will include a *Startup.cs* file that is responsible for setting up request middleware in a way that's very similar to how ASP.NET Core behaves.

If you need to run code when your ASP.NET MVC app starts up, it will typically use one of these approaches.

ASP.NET Core Startup

As noted previously, ASP.NET Core apps are standalone programs. As such, they typically include a *Program.cs* file containing the entry point for the app. A typical example of this file is shown in Figure 2-1.

```csharp
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

Figure 2-1. A typical ASP.NET Core Program.cs file.

The code shown in Figure 2-1 creates a *host* for the app, builds it, and runs it. The ASP.NET Core app runs within the host configured by the IHostBuilder shown. While it's possible to completely configure an ASP.NET Core app using the IHostBuilder, typically the bulk of this work is done in a Startup class.

The Startup class exposes two methods to the host: ConfigureServices and Configure. The ConfigureServices method is used to define the services the app will use and their respective lifetimes. The Configure method is used to define how each request to the app will be handled by setting up a request pipeline composed of middleware.

In addition to code related to configuring the app's services or request pipeline, apps may have other code that must run when the app begins. Such code is typically placed in *Program.cs* or registered as an IHostedService, which will be started by the generic host when the app starts.

The IHostedService interface just exposes two methods, StartAsync and StopAsync. You register the interface in ConfigureServices and the host does the rest, calling the StartAsync method before the app starts up.

Porting considerations

Teams looking to migrate their apps from ASP.NET MVC to ASP.NET Core need to identify what code is being run when their app starts up and determine the appropriate location for such code in their ASP.NET Core app. For custom code needed to run when the app starts up, especially async code, the recommended approach is typically to put such code into IHostedService implementations.

References

- ASP.NET Application Life Cycle Overview for IIS 7
- ASP.NET Application Life Cycle Overview for IIS 5 and 6
- Getting Started with OWIN and Katana
- WebActivator
- App Startup in ASP.NET Core
- .NET Generic Host in ASP.NET Core
- IHostedService

Hosting differences between ASP.NET MVC and ASP.NET Core

ASP.NET MVC apps are hosted in IIS, and may rely on IIS configuration for their behavior. This often includes IIS modules. As part of reviewing an app to prepare to port it from ASP.NET MVC to ASP.NET Core, teams should identify which modules, if any, they're using from the list of IIS Modules installed on their server.

ASP.NET Core apps can run on a number of different servers. The default cross platform server, Kestrel, is a good default choice. Apps running in Kestrel can be hosted by IIS, running in a separate process. On Windows, apps can also run in process on IIS or using HTTP.sys. Kestrel is generally recommended for best performance. HTTP.sys can be used in scenarios where the app is exposed to the Internet and required capabilities are supported by HTTP.sys but not Kestrel.

Kestrel does not have an equivalent to IIS modules (though apps hosted in IIS can still take advantage of IIS modules). To achieve equivalent behavior, middleware configured in the ASP.NET Core app itself is typically used.

References

- [IIS Modules](#)
- [ASP.NET Core Servers](#)

Serve static files in ASP.NET MVC and ASP.NET Core

Most web apps involve a combination of server-side logic and static files that must be sent to the client as-is. How should your migration from ASP.NET MVC to ASP.NET Core handle serving static files?

Host static files in ASP.NET MVC

ASP.NET MVC apps, hosted by IIS, typically host static files directly from the app. ASP.NET MVC supports placing static files side by side with files that should be kept private on the server. IIS and ASP.NET require explicitly restricting certain files or file extensions from being served from the folder in which an ASP.NET app is hosted.

For many static files, using a content delivery network (CDN) is a good practice. [Static content hosting](#) allows better performance while reducing load and bandwidth from app servers.

Host static files in ASP.NET Core

It may be surprising, but ASP.NET Core doesn't have built-in support for static files. This feature that has always existed as just a part of ASP.NET, enabled by IIS, isn't intrinsic to ASP.NET Core or its Kestrel web server. To serve static files from an ASP.NET Core app, you must configure [static files middleware](#).

With static files middleware configured, an ASP.NET Core app will serve all files located in a certain folder (typically */wwwroot*). No other files in the app or project folder are at risk of being accidentally exposed by the server. No special restrictions based on file names or extensions need to be configured, as is the case with IIS. Instead, developers explicitly choose to expose files publicly when they place them in the *wwwroot* folder. By default, files outside of this folder aren't shared.

Because support for static files uses middleware, any other middleware can be applied as part of the same request pipeline. Examples of middleware include authentication, caching, and compression.

Of course, CDNs remain a good choice for ASP.NET Core apps for all the same reasons they're used in ASP.NET MVC apps. As part of preparing to migrate to .NET Core, if there are benefits your app could realize from using a CDN, it would be good to move static files to a CDN before migrating to .NET Core. Doing so reduces the migration effort's overall scope for static assets.

References

- [Static content hosting](#)
- [Static files in ASP.NET Core](#)

Dependency injection differences between ASP.NET MVC and ASP.NET Core

Although dependency injection (DI) isn't built into ASP.NET MVC or Web API, many apps enable it by adding a NuGet package with an inversion of control (IOC) container. These are sometimes referred to as DI containers, for dependency injection (or inversion). Some of the most popular containers used in ASP.NET MVC apps include:

- Autofac
- Unity
- Ninject
- StructureMap *(deprecated)*
- Castle Windsor

If your ASP.NET MVC app isn't using DI, you will probably want to investigate the built-in support for DI in ASP.NET Core. DI promotes loose coupling of modules in your app and enables better testability and adherence to principles like SOLID.

If your app does use DI, then probably your best course of action is to see if the container you're using supports ASP.NET Core. If so, you may be able to continue using it and your custom configuration rules describing your type mappings and lifetimes.

Either way, you should consider using the built-in support for DI that ships with ASP.NET Core, as it may meet your app's needs.

Dependency injection in ASP.NET Core

ASP.NET Core assumes apps will use DI. It's not just built into the framework, but is required in order to bring support for framework features into your ASP.NET Core apps. In app startup, a call is made to ConfigureServices which is responsible for registering all of the types that the DI container (service collection/service provider) can create and inject in the app. Built-in ASP.NET Core features like Entity Framework Core, Identity, and even MVC are brought into the app by configuring them as services in the ConfigureServices method.

In addition to using the default implementation, apps can still use custom containers. The documentation covers how to replace the default service container.

DI is fundamental to ASP.NET Core. If your team isn't already well-versed in this practice, you'll want to understand it before porting your app.

References
- Dependency Injection in .NET
- Dependency Injection in ASP.NET Core

Compare middleware to modules and handlers

If your existing ASP.NET MVC or Web API app uses OWIN/Katana, you're most likely already familiar with the concept of middleware and porting it to ASP.NET Core should be fairly straightforward. However, most ASP.NET apps rely on HTTP modules and HTTP handlers instead of middleware. Migrating these to ASP.NET Core requires extra effort.

ASP.NET modules and handlers

HTTP modules and HTTP handlers are an integral part of the ASP.NET architecture. While a request is being processed, each request is processed by multiple HTTP modules (for example, the authentication module and the session module) and is then processed by a single HTTP handler. After the handler has processed the request, the request flows back through the HTTP modules.

If your app is using custom HTTP modules or HTTP handlers, you'll need a plan to migrate them to ASP.NET Core. The most likely replacement in ASP.NET Core is middleware.

ASP.NET Core middleware

ASP.NET Core defines a request pipeline in each app's Configure method. This request pipeline defines how an incoming request is handled by the app, with each method in the pipeline calling the next method until eventually a method terminates, and the chain of *middleware* terminates and returns back up the stack. Middleware can target all requests, or can be configured to only map to certain requests based on the requested path or other factors. It can be configured wholly in the Configure method of an app, or implemented in a separate class.

Behavior in an ASP.NET MVC app that uses HTTP modules is probably best suited to custom middleware. Custom HTTP handlers can be replaced with custom routes or endpoints that respond to the same path.

References

- ASP.NET HTTP modules and HTTP handlers
- ASP.NET Core middleware

Configuration differences between ASP.NET MVC and ASP.NET Core

How configuration values are stored and read changed dramatically between ASP.NET and ASP.NET Core.

ASP.NET MVC configuration

In ASP.NET apps, configuration uses the built-in .NET configuration files, *web.config* in the app folder and *machine.config* on the server. Most ASP.NET MVC and Web API apps store their settings in the

configuration file's appSettings or connectionStrings elements. Some also use custom configuration sections that can be mapped to a settings class.

Configuration in a .NET Framework app is accessed using the System.Configuration.ConfigurationManager class. Unfortunately, this class provides static access to the configuration elements. As a result, very few ASP.NET MVC apps tend to abstract access to their configuration settings or inject them where needed. Instead, most .NET Framework apps tend to directly access the configuration system anywhere the app needs to use a setting.

Typical ASP.NET MVC configuration access:

```
string connectionString =
    ConfigurationManager.ConnectionStrings["DefaultConnection"]
        .ConnectionString;
```

ASP.NET Core configuration

In ASP.NET Core apps, configuration is, itself, configurable. However, most apps use a set of defaults provided as part of the standard project templates and the ConfigureWebHostDefaults method used in them. The default settings use JSON formatted files, with the ability to override settings in the base *appsettings.json* file with environment-specific files like *appsettings.Development.json*. Additionally, the default configuration system further overrides all file-based settings with any environment variable that exists for the same named setting. This is useful in many scenarios and is especially useful when deploying to a hosting environment, since it eliminates the need to worry about whether deploying configuration files will accidentally overwrite important production configuration settings. Configuration values can also be provided as command line arguments.

Accessing configuration values can be done in many ways in .NET Core. Because dependency injection is built into .NET Core, configuration values are generally accessed through an interface that is injected into classes that need them. This can involve passing a interface like IConfiguration, but usually it's better to pass just the settings required by the class using the options pattern.

Figure 2-2 shows how to pass IConfiguration into a Razor Page and access configuration settings from it:

```
using Microsoft.Extensions.Configuration;

public class TestModel : PageModel
{
    private readonly IConfiguration _configuration;

    public TestModel(IConfiguration configuration)
    {
        _configuration= configuration;
    }

    public ContentResult OnGet()
    {
        var myKeyValue = _configuration["MyKey"];

        // ...
    }
}
```

Using the options pattern, settings access is similar but is strongly typed and more specific to the setting(s) needed by the consuming class, as Figure 2-3 demonstrates.

```csharp
public class PositionOptions
{
    public const string Position = nameof(Position);

    public string Title { get; set; }
    public string Name { get; set; }
}

public class Test2Model : PageModel
{
    private readonly PositionOptions _options;

    public Test2Model(IOptions<PositionOptions> options)
    {
        _options = options.Value;
    }

    public ContentResult OnGet()
    {
        return Content($"Title: {_options.Title}\nName: {_options.Name}");
    }
}
```

Figure 2-3. Using the options pattern in ASP.NET Core.

For the options pattern to work, the options type must be configured in ConfigureServices when the app starts up:

```csharp
// required in ConfigureServices
services.Configure<PositionOptions>(Configuration.GetSection(PositionOptions.Position));
```

Migrate configuration

When considering how to port an app's configuration settings from .NET Framework to .NET Core, the first step is to identify all of the configuration settings that are being used. Most of these will be in the *web.config* file in the app's root folder, but some apps expect settings to be found in the shared *machine.config* file as well.

Once all settings in the config files have been cataloged, the next step should be to identify where and how the settings are used in the app itself. If some settings aren't being used, these can probably be omitted from the migration. For each setting, note all of the places it's being used so you can be sure you don't miss any when you migrate the code.

If you're still maintaining the ASP.NET app, it may be helpful to avoid static references to ConfigurationManager and replace them with access through interfaces. This will ease the transition to ASP.NET Core's configuration system. In general, static access to external resources makes code harder to test and maintain, so be on the lookout for anywhere else the app may be following this pattern.

References

Routing differences between ASP.NET MVC and ASP.NET Core

Routing is responsible for mapping incoming browser requests to particular controller actions (or Razor Pages handlers). This section covers how routing differs between ASP.NET MVC (and Web API) and ASP.NET Core (MVC, Razor Pages, and otherwise).

Routing in ASP.NET MVC and Web API

ASP.NET MVC offers two approaches to routing:

1. The route table, which is a collection of routes that can be used to match incoming requests to controller actions.
2. Attribute routing, which performs the same function but is achieved by decorating the actions themselves, rather than editing a global route table.

Route table

The route table is configured when the app starts up. Typically, a static method call is used to configure the global route collection, like so:

```csharp
public class MvcApplication : System.Web.HttpApplication
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
        routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new { controller = "Home", action = "Index", id = "" },
            constraints: new { id = "\\d+" }
        );
    }

    protected void Application_Start()
    {
        RegisterRoutes(RouteTable.Routes);
    }
}
```

In the preceding code, the route table is managed by the RouteCollection type, which is used to add new routes with MapRoute. Routes are named and include a route string, which can include parameters for controllers, actions, areas, and other placeholders. If an app follows a standard

convention, most of its actions can be handled by this single default route, with any exceptions to this convention configured using additional routes.

Attribute routing in ASP.NET MVC

Routes that are defined with their actions may be easier to discover and reason about than routes defined in an external location. Using attribute routing, an individual action method can have its route defined with a [Route] attribute:

```csharp
public class ProductsController
{
    [Route("products")]
    public ActionResult Index()
    {
        return View();
    }

    [Route("products/{id}")]
    public ActionResult Details(int id)
    {
        return View();
    }
}
```

Attribute routing in ASP.NET MVC 5 also supports defaults and prefixes, which can be added at the controller level (and which are applied to all actions within that controller). Refer to the documentation for details.

Setting up attribute routing requires adding one line to the default route table configuration:

```csharp
routes.MapMvcAttributeRoutes();
```

Attribute routing can take advantage of route constraints, both built-in and custom, and supports named routes and areas using the [RouteArea] attribute. If your app uses areas, you'll need to configure support for them in your route registration code by adding one more line:

```csharp
routes.MapMvcAttributeRoutes();

AreaRegistration.RegisterAllAreas();
```

Attribute routing in ASP.NET Web API 2

Attribute routing in ASP.NET Web API 2 is similar to routing in ASP.NET MVC 5, with minor differences. Configuring Web API 2 is typically done in its own class, which is called during app startup. Attribute routing configuration is handled in this class:

```csharp
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        // Attribute routing.
        config.MapHttpAttributeRoutes();

        // Convention-based routing.
        config.Routes.MapHttpRoute(
```

```
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional }
        );
    }
}
```

As shown in the preceding code, attribute routing may be combined with convention-based routing in Web API apps.

In addition to attribute routing, ASP.NET Web API chooses which action to call based on the HTTP method (for example, GET or POST), the {action} placeholder in a route (if any), and parameters of the action. In many cases, the name of the action will help determine whether it's matched, since prefixing the action name with "Get" or "Post" is used to match the appropriate HTTP method to it. Alternatively, actions can be decorated with an appropriate HTTP method attribute, like [HttpGet], allowing the use of action names that aren't prefixed with an HTTP method.

```
public class ProductsController : ApiController
{
    // matched by name and (lack of) parameters
    public IEnumerable<Product> GetAll() { }

    // matched by GET and string parameter
    [HttpGet]
    public IEnumerable<Product> FindProductsByName(string name) { }
}
```

Given the above controller, an HTTP GET request to localhost:123/products/ matches the GetAll action. An HTTP GET request to localhost:123/products?name=ardalis matches the FindProductsByName action.

Routing in ASP.NET Core 3.1

In ASP.NET Core, routing is handled by routing middleware, which matches the URLs of incoming requests to actions or other endpoints. Controller actions are either conventionally routed or attribute-routed. Conventional routing is similar to the route table approach used in ASP.NET MVC and Web API. Whether you're using conventional, attribute, or both, you need to configure your app to use the routing middleware. To use the middleware, add the following code to your Startup.Configure method:

```
app.UseRouting();
```

Conventional routing

With conventional routing, you set up one or more conventions that will be used to match incoming URLs to *endpoints* in the app. In ASP.NET Core, these endpoints may be controller actions, like in ASP.NET MVC or Web API. The endpoints could also be Razor Pages, Health Checks, or SignalR hubs. All of these routable features are configured in a similar fashion using endpoints:

```
// in Startup.Configure()
app.UseEndpoints(endpoints =>
{
    endpoints.MapHealthChecks("/healthz").RequireAuthorization();
```

```
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
    endpoints.MapRazorPages();
});
```

The preceding code is used (in addition to UseRouting) to configure various endpoints, including Health Checks, controllers, and Razor Pages. For controllers, the above configuration specifies a default routing convention, which is the fairly standard {controller}/{action}/{id?} pattern that's been recommended since the first versions of ASP.NET MVC.

Attribute routing

Attribute routing in ASP.NET Core is the preferred approach for configuring routing in controllers. If you're building APIs, the [ApiController] attribute should be applied to your controllers. Among other things, this attribute requires the use of attribute routing for actions in such controller classes.

Attribute routing in ASP.NET Core behaves similarly in ASP.NET MVC and Web API. In addition to supporting the [Route] attribute, however, route information can also be specified as part of the HTTP method attribute:

```
[HttpGet("api/products/{id}")]
public async ActionResult<Product> Details(int id)
{
    // ...
}
```

As with previous versions, you can specify a default route with placeholders, and add this at the controller class level or even on a base class. You use the same [Route] attribute for all of these cases. For example, a base API controller class might look like this:

```
[Route("api/{controller}/{action}/{id?:int}")]
public abstract class BaseApiController : ControllerBase, IApiController
{
    // ...
}
```

Using this attribute, classes inheriting from this type would route URLs to actions based on the controller name, action name, and an optional integer id parameter.

References

- ASP.NET MVC Routing Overview
- Attribute Routing in ASP.NET MVC 5
- Attribute routing in ASP.NET Web API 2
- Routing and Action Selection in ASP.NET Web API
- Routing in ASP.NET Core
- Routing to controller actions in ASP.NET Core MVC

Compare Razor Pages to ASP.NET MVC

Razor Pages is the preferred way to create page- or form-based apps in ASP.NET Core. From the docs, "Razor Pages can make coding page-focused scenarios easier and more productive than using controllers and views." If your ASP.NET MVC app makes heavy use of views, you may want to consider migrating from actions and views to Razor Pages.

A typical strongly typed view-based MVC app will use a controller to contain one or more actions. The controller will interact with the domain or data model, and create an instance of a viewmodel class. Then this viewmodel class is passed to the view associated with that action. Using this approach, coupled with the default folder structure of MVC apps, to add a new page to an app requires modifying a controller in one folder, a view in a nested subfolder in another folder, and a viewmodel in yet another folder.

Razor Pages group together the action (now a *handler*) and the viewmodel (called a *PageModel*) in one class, and link this class to the view (called a Razor Page). All Razor Pages go into a *Pages* folder in the root of the ASP.NET Core project. Razor Pages use a routing convention based on their name and location within this folder. Handlers behave exactly like action methods but have the HTTP verb they handle in their name (for example, OnGet). They also don't necessarily need to return, since by default they're assumed to return the page they're associated with. This tends to keep Razor Pages and their handlers smaller and more focused while at the same time making it easier to find and work with all of the files needed to add or modify a particular part of an app.

As part of a move from ASP.NET MVC to ASP.NET Core, teams should consider whether they want to migrate controllers and views to ASP.NET Core controllers and views, or to Razor Pages. The former will most likely require slightly less overall effort, but won't allow the team to take advantage of the benefits of Razor Pages over traditional view-based file organization.

References

- Introduction to Razor Pages in ASP.NET Core
- Simpler ASP.NET Core Apps with Razor Pages

Compare ASP.NET Web API 2 and ASP.NET Core

ASP.NET Core offers iterative improvements to ASP.NET Web API 2, but should feel familiar to developers who have used Web API 2. ASP.NET Web API 2 was developed and shipped alongside ASP.NET MVC. This meant the two approaches had similar-but-different approaches to things like attribute routing and dependency injection. In ASP.NET Core, there's no longer any distinction between MVC and Web APIs. There's only ASP.NET MVC, which includes support for view-based scenarios, API endpoints, and Razor Pages (and other variations like health checks and SignalR).

In addition to being consistent and unified within ASP.NET Core, APIs built in .NET Core are much easier to test than those built on ASP.NET Web API 2. We'll cover testing differences in more detail in a moment. The built-in support for hosting ASP.NET Core apps, in a test host that can create an

HttpClient that makes in-memory requests to the app, is a huge benefit when it comes to automated testing.

When migrating from ASP.NET Web API 2 to ASP.NET Core, the transition is straightforward. If you have large, bloated controllers, one approach you may consider to break them up is the use of the Ardalis.ApiEndpoints NuGet packages. This package breaks up each endpoint into its own specific class, with associated request and response types as appropriate. This approach yields many of the same benefits as Razor Pages offer over view-based code organization.

References

- Migrate from ASP.NET Web API to ASP.NET Core
- Ardalis.ApiEndpoints NuGet package

Compare authentication and authorization between ASP.NET MVC and ASP.NET Core

In ASP.NET MVC 5, authentication is configured in *Startup.Auth.cs* in the *App_Start* folder. In ASP.NET Core MVC, this configuration occurs in Startup.cs. Authentication and authorization are performed using middleware added to the request pipeline in ConfigureServices:

```
// inside Startup.ConfigureServices

app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
    endpoints.MapRazorPages();
});
```

It's important to add the auth middleware in the appropriate order in the middleware pipeline. Only requests that make it to the middleware will be impacted by it. For instance, if a call to UseStaticFiles() was placed above the code shown here, it wouldn't be protected by authentication and authorization.

In ASP.NET MVC and Web API, apps often refer to the current user using the ClaimsPrincipal.Current property. This property isn't set in ASP.NET Core, and any behavior in your app that depends on it will need to migrate from ClaimsPrincipal.Current by using the User property on ControllerBase or getting access to the current HttpContext and referencing its User property. If neither of these solutions is an option, services can request the User as an argument, in which case it must be supplied from elsewhere in the app, or the IHttpContextAccessor can be requested and used to get the HttpContext.

Authorization

Authorization defines what a given user can do within the app. It's separate from authentication, which is concerned merely with identifying who the user is. ASP.NET Core provides a simple, declarative role and a rich, policy-based model for authorization. Specifying that a resource requires authorization is often as simple as adding the [Authorize] attribute to the action or controller. If you're migrating to Razor Pages from MVC views, you should specify conventions for authorization when you configure Razor Pages in Startup.

Authorization in ASP.NET Core may be as simple as prohibiting anonymous users while allowing authenticated users. Or it can scale up to support role-based, claims-based, or policy-based authorization approaches. For more information on these approaches, see the documentation on authorization in ASP.NET Core. You'll likely find that one of them is closely aligned with your current authorization approach.

References

- Security, Authentication, and Authorization with ASP.NET MVC
- Migrate Authentication and Identity to ASP.NET Core
- Migrate from ClaimsPrincipal.Current
- Introduction to Authorization in ASP.NET Core

Compare ASP.NET Identity and ASP.NET Core Identity

In ASP.NET MVC, identity features are typically configured in *IdentityConfig.cs* in the *App_Start* folder. Review how this is configured in the existing app, and compare it to the configuration required for ASP.NET Core Identity in *Startup.cs*.

ASP.NET Identity is an API that supports user interface login functionality and manages users, passwords, profile data, roles, claims, tokens, email confirmations, and more. It supports external login providers like Facebook, Google, Microsoft, and Twitter.

If your ASP.NET MVC app is using ASP.NET Membership, you'll find a guide to migrate from ASP.NET Membership authentication to ASP.NET Core 2.0 Identity. This is mainly a data migration exercise, at the completion of which you should be able to use ASP.NET Core Identity with your migrated user records.

If you migrate your ASP.NET Identity users to ASP.NET Core Identity, you may need to update their password hashes, or track which passwords are hashed with the new ASP.NET Core Identity approach or the older ASP.NET Identity approach. This approach described on Stack Overflow provides some options for migrating user password hashes over time, rather than all at once.

One of the biggest differences when it comes to ASP.NET Core Identity compared to ASP.NET Identity is how little code you need to include in your project. ASP.NET Core Identity now ships as a Razor Class Library, meaning its UI and logic are all available from a NuGet package. You can override the

existing UI and logic by scaffolding the ASP.NET Core Identity files but even in this case you need only scaffold the pages you want to modify, not every one that exists.

Migrate from OWIN / Katana

The following resources offer some guidance for migrating from OWIN / Katana:

- Migration
- Katana to ASPNET 5

References

- Migrate Authentication and Identity to ASP.NET Core
- Introduction to Identity on ASP.NET Core
- Configure ASP.NET Core Identity
- Scaffold Identity in ASP.NET Core projects

Compare controllers in ASP.NET MVC and Web API with controllers in ASP.NET Core

In ASP.NET MVC 5 and Web API 2, there were two different Controller base types. MVC controllers inherited from Controller; Web API controllers inherited from ApiController. In ASP.NET Core, this object hierarchy has been merged. It's recommended that API controllers in ASP.NET Core inherit from ControllerBase and add the [ApiController] attribute. Standard view-based MVC controllers should inherit from Controller.

In both frameworks, controllers are used to organize sets of action methods. Filters and routes can be applied on a controller level in addition to at the action level. These conventions can be extended further by using custom base Controller types with default behavior and attributes applied to them.

In ASP.NET MVC, content negotiation isn't supported. ASP.NET Web API 2 does support content negotiation, as does ASP.NET Core. Using content negotiation, the format of the content returned to a request can be determined by headers the client provides indicating its preferred manner of receiving the content.

When migrating ASP.NET Web API controllers to ASP.NET Core, a few components need to be changed if they exist. These include references to the ApiController base class, the System.Web.Http namespace, and the IHttpActionResult interface. Refer to the documentation for recommendations on how to migrate these specific differences. Note that the preferred return type for API actions in ASP.NET Core is ActionResult<T>.

In addition, the [ChildActionOnly] attribute isn't supported in ASP.NET Core. In ASP.NET Core, similar functionality is achieved using View Components.

ASP.NET Core includes two new attributes: ConsumesAttribute and ProducesAttribute. These are used to specify the type an action consumes or produces, which can be helpful for routing and documenting the API using tools like Swagger/OpenAPI.

References

- [Format response data in ASP.NET Core Web API](#)
- [Migrate from ASP.NET Web API to ASP.NET Core](#)

Compare Razor usage in ASP.NET MVC and ASP.NET Core

The basic syntax of Razor hasn't changed substantially between ASP.NET MVC and ASP.NET Core. However, there are certain differences, such as the introduction of [Tag Helpers](#) and Razor Pages, that should be considered when migrating. If your app makes heavy use of custom Razor functionality, refer to the [Razor syntax reference for ASP.NET Core](#) to see what changes may be required when you migrate to ASP.NET Core.

Tag Helpers

Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. They offer many advantages over HTML Helpers and should be used in place of HTML Helpers wherever possible. They provide an HTML-friendly development experience, since they look like standard HTML and are ignored by most tooling designed to edit HTML. Within *Visual Studio*, there's rich IntelliSense support for creating HTML and Razor markup with Tag Helpers. Tag Helpers can provide simple or complex behavior from declarative markup in Razor files.

Razor Pages

Razor Pages offer an alternative to controllers, actions, and views for page- and form-based apps. [Razor Pages were compared to ASP.NET MVC earlier in this chapter](#).

References

- [Migrate from ASP.NET MVC to ASP.NET Core MVC: Controllers and Views](#)
- [Tag Helpers in ASP.NET Core](#)
- [Introduction to Razor Pages in ASP.NET Core](#)
- [Razor syntax reference for ASP.NET Core](#)

Compare ASP.NET SignalR and ASP.NET Core SignalR

ASP.NET Core SignalR is incompatible with clients or servers using ASP.NET SignalR. You'll need to update both clients and server to use ASP.NET Core SignalR. Some differences are described in this section, while the full list is available in the [docs](#). ASP.NET Core SignalR requires .NET Core 2.1 or greater.

Feature differences

- ASP.NET SignalR automatically attempts to reconnect dropped connections; this behavior is opt-in for ASP.NET Core SignalR clients
- Both frameworks support JSON; ASP.NET Core SignalR also supports a binary protocol based on MessagePack, and custom protocols can be created.
- The Forever Frame transport, supported by ASP.NET SignalR, isn't supported in ASP.NET Core SignalR.
- ASP.NET Core SignalR must be configured by adding services.AddSignalR() and app.UseEndpoints in Startup.ConfigureServices and Startup.Configure, respectively.
- ASP.NET Core SignalR requires sticky sessions; ASP.NET SignalR doesn't.
- ASP.NET Core simplifies the connection model; connections are only made to a single hub.
- ASP.NET Core SignalR supports streaming data from the hub to the client.
- ASP.NET Core SignalR doesn't support passing state between clients and the hub.
- The PersistentConnection class doesn't exist in ASP.NET Core SignalR.
- ASP.NET SignalR supports SQL Server and Redis. ASP.NET Core SignalR supports Azure SignalR and Redis.

References

- Differences between ASP.NET SignalR and ASP.NET Core SignalR
- Azure SignalR Service

Compare testing options between ASP.NET MVC and ASP.NET Core

ASP.NET MVC apps support unit testing of controllers, but this approach often omits many MVC features like routing, authorization, model binding, model validation, filters, and more. To test these MVC features in addition to the logic within the controller action itself, frequently the app would need to be deployed and then tested with a tool like Selenium. These tests are substantially more expensive, more brittle, and slower than typical unit tests that can be run without the need for hosting and running the entire app.

ASP.NET Core controllers can be unit tested just like ASP.NET MVC controllers, but with the same limitations. However, ASP.NET Core supports fast, easy-to-author integration tests as well. Integration tests are hosted by a TestHost class and are typically configured in a custom WebApplicationFactory that can override or replace app dependencies. For instance, frequently during integration tests the app will target a different data source and may replace services that send emails with fake or mock implementations.

ASP.NET MVC and Web API did not support anything like the integration testing scenarios available in ASP.NET Core. In any migration effort, you should allocate time to write some integration tests for your newly migrated system to ensure it's working as expected and continues to do so. Even if you

weren't writing tests of your web app logic before the migration, you should strongly consider doing so as you move to ASP.NET Core.

References

- [Creating Unit Tests for ASP.NET MVC Applications](#)
- [Unit test controller logic in ASP.NET Core](#)
- [Integration tests in ASP.NET Core](#)

Migrate large solutions to ASP.NET Core

Migrating large solutions from .NET Framework to .NET Core requires some planning. Dependencies must be migrated or updated before the projects that depend on them. There are tools that can identify dependencies and offer help with migrating to .NET Core. Depending on the app, its scope, and current usage patterns, different strategies may be employed when migrating. Do you migrate it all at once, or over time, side-by-side with the current system? Do you wrap the current system in the new one, and incrementally replace its functionality?

In this chapter, you'll learn how create a migration plan for a large solution, how to use tools to help with the migration, and some strategies to consider for the migration itself.

References

- [What topics are important to migrating large MVC and Web API apps to .NET Core?](#)
- [Porting from .NET Framework to .NET Core](#)

Identify sequence of projects to migrate

For solutions that involve multiple front-end apps, it's best to migrate the apps one by one. For example, create a solution that only includes one front-end app and its dependencies so you can easily identify the scope of work involved. Solutions are lightweight, and you can include projects in multiple solutions if needed. Take advantage of solutions as an organizational tool when migrating.

Once you've identified the ASP.NET app to migrate and have its dependent projects located with it (ideally in a solution), the next step is to identify framework and NuGet dependencies. Having identified all dependencies, the simplest migration approach is a "bottom up" approach. With this approach, the lowest level of dependencies is migrated first. Then the next level of dependencies is migrated, until eventually the only thing left is the front-end app. Figure 3-1 shows an example set of projects composing an app. The low-level class libraries are at the bottom, and the ASP.NET MVC project is at the top.
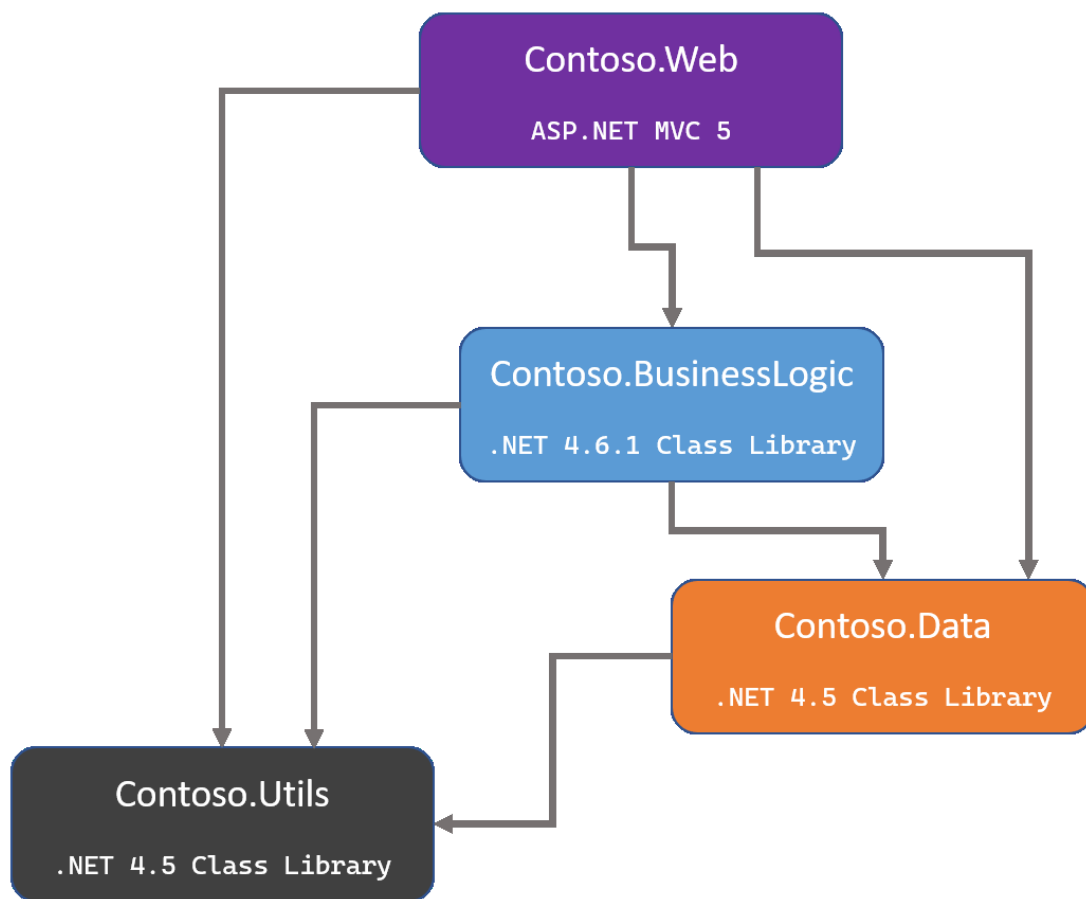
Figure 3-1. Project dependencies graph.

Choose a particular front-end app, an ASP.NET MVC 5 / Web API 2 project. Identify its dependencies in the solution, and map out their dependencies until you have a complete list. A diagram like the one shown in Figure 3-1 may be useful when mapping out project dependencies. Visual Studio can produce a dependency diagram for your solution, depending on which edition you're using. The .NET Portability Analyzer can also produce dependency diagrams.

Figure 3-2 shows the installer for the .NET Portability Analyzer Visual Studio extension:
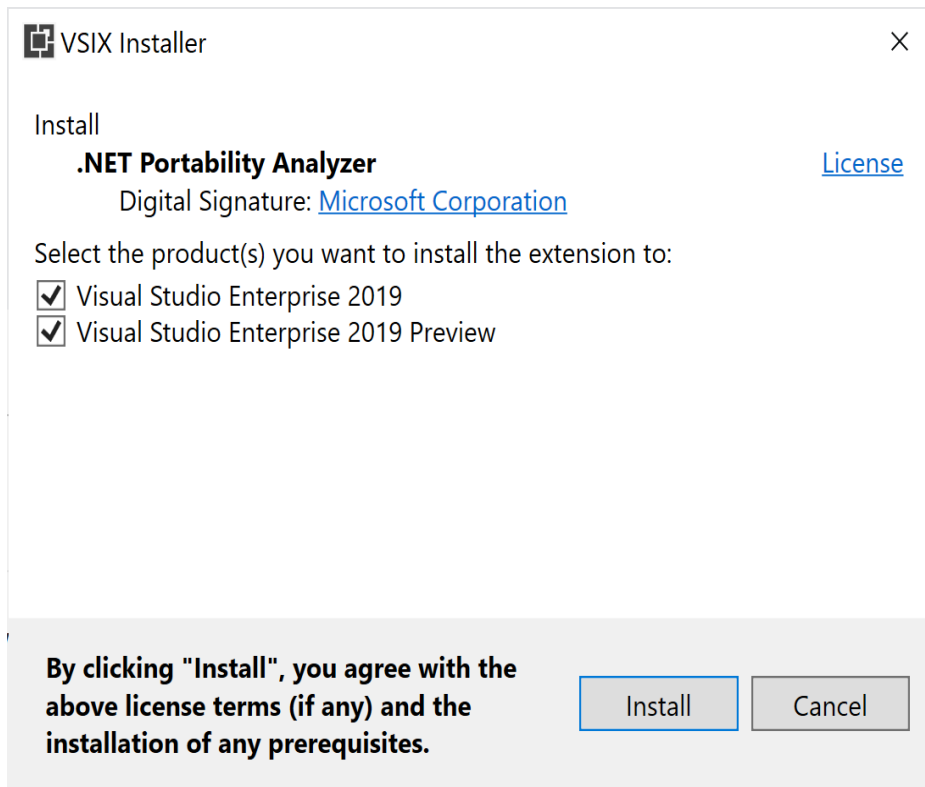
Figure 3-2. .NET Portability Analyzer installer.

The extension supports Visual Studio 2017 and later. Once installed, you configure it from the **Analyze** > **Portability Analyzer Settings** menu, as shown in Figure 3-3.
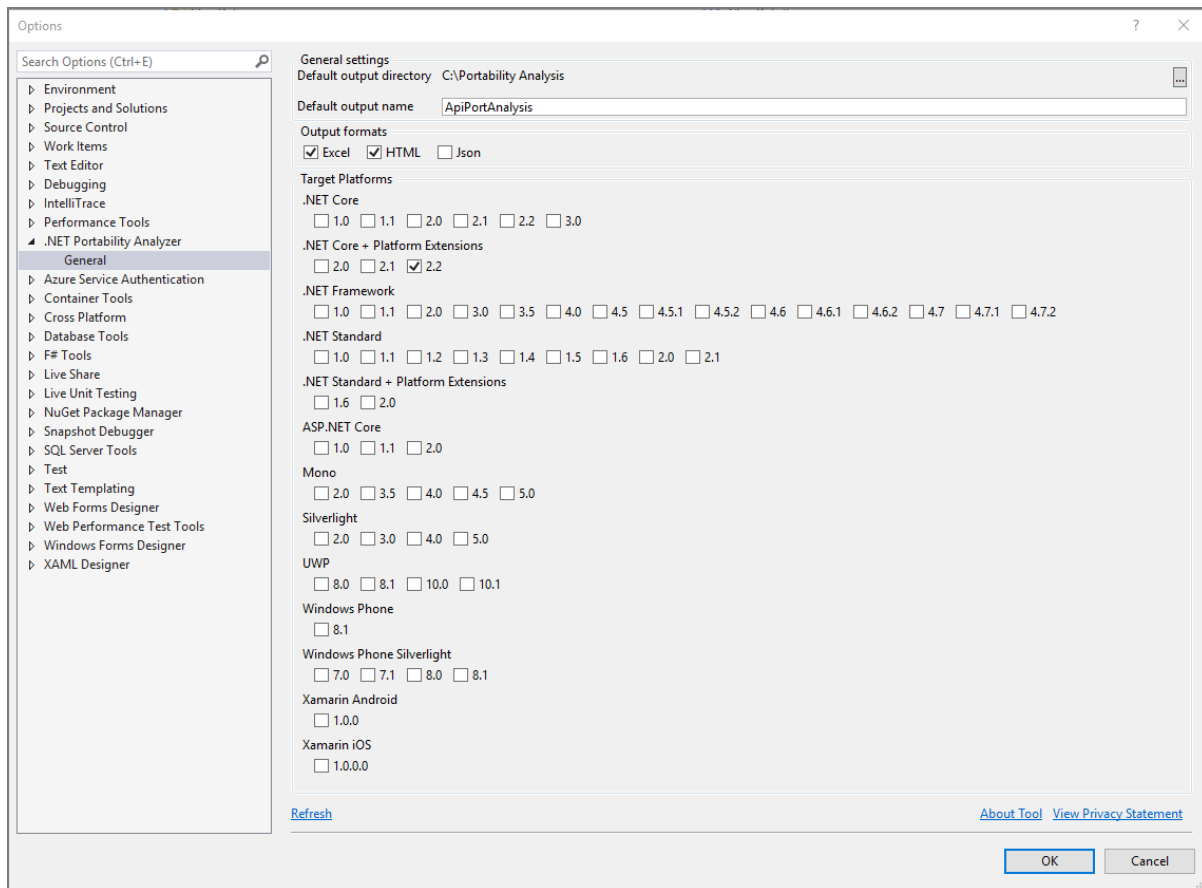
Figure 3-3. Configure the .NET Portability Analyzer.

The analyzer produces a detailed report for each assembly. The report:

- Describes how compatible each project is with a given target framework, such as .NET Core 3.1 or .NET Standard 2.0.
- Helps teams assess the effort required to port a particular project to a particular target framework.

The details of this analysis are covered in the next section.

Once you've mapped out the projects and their relationships with one another, you're ready to determine the order in which you'll migrate the projects. Begin with projects that have no dependencies. Then, work your way up the tree to the projects that depend on these projects.

In the example shown in Figure 3-1, you would start with the *Contoso.Utils* project, since it doesn't depend on any other projects. Next, *Contoso.Data* since it only depends on "Utils". Then migrate the "BusinessLogic" library, and finally the front-end ASP.NET "Web" project. Following this "bottom up" approach works well for relatively small and well-factored apps that can be migrated as a unit once all of their projects have migrated. Larger apps with more complexity, or more code that will take longer to migrate, may need to consider more incremental strategies.

Unit tests

Missing from the previous diagrams are unit test projects. Hopefully there are tests covering at least some of the existing behavior of the libraries being ported.

If you have unit tests, it's best to convert those projects first. You'll want to continue testing changes in the project you're working on. Remember that porting to .NET Core is a significant change to your codebase.

MSTest, xUnit, and NUnit all work on .NET Core. If you don't currently have tests for your app, consider building some characterization tests that verify the system's current behavior. The benefit is that once the migration is complete, you can confirm the app's behavior remains unchanged.

Considerations for migrating many apps

Some organizations will have many different apps to migrate, and migrating each one by hand may require too many resources to be tenable. In these situations, some degree of automation is recommended. The steps followed in this chapter can be automated. Structural changes, like project file differences and updates to common packages, can be applied by scripts. These scripts can be refined as they're run iteratively on more projects. On each run, examine whatever manual steps are required for each project. Automate those manual steps, if possible. Using this approach, the organization should grow faster and better at porting their apps over time, with improved automation support each step of the way.

Watch an overview of how to employ this approach in this [dotNetConf presentation by Lizzy Gallagher of Mastercard](). The five phases employed in this presentation included:

- Migrate third-party NuGet dependencies
- Migrate apps to use new *.csproj* file format
- Migrate apps to ASP.NET Core (targeting .NET Framework)
- Update internal NuGet dependencies to .NET Standard
- Update all apps to target .NET Core 3.1

When automating a large suite of apps, it helps significantly if they follow consistent coding guidelines and project organization. Automation efforts rely on this consistency to be effective. In addition to parsing and migrating project files, common code patterns can be migrated automatically. Some code pattern examples include differences in how controller actions are declared or how they return results.

For example, a migration script could search files matching *Controller.cs* for lines of code matching one of these patterns:

```
return new HttpStatusCodeResult(200);
// or
return new HttpStatusCodeResult(HttpStatusCode.OK);
```

In ASP.NET Core, either of the preceding lines of code can be replaced with:

```
return Ok();
```

Summary

The best approach to porting a large .NET Framework app to .NET Core is to:

1. Identify project dependencies.
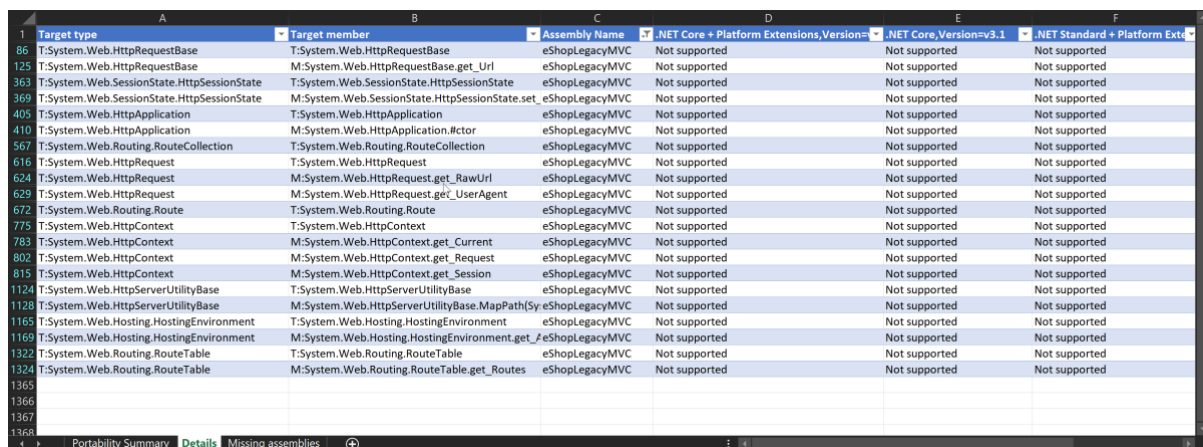2. Analyze what's required to port each project.
3. Start from the bottom up.

You can use the .NET Portability Analyzer to determine how compatible existing libraries may be with target platforms. Automated tests will help ensure no breaking changes are introduced as the app is ported. These test projects should be among the first projects ported.

References

- [Porting from .NET Framework to .NET Core](#)
- [The .NET Portability Analyzer](#)
- [Channel 9: A Brief Look at the .NET Portability Analyzer (Video)](#)
- [2 Years, 200 Apps: A .NET Core Migration at Scale (Video)](#)

Understand and update dependencies

After identifying the sequence in which the app's individual projects must be migrated, the next step is to understand each project's dependencies and update them if necessary. For platform dependencies, the best way to start is to run the .NET Portability Analyzer on the assembly in question, and then look at the detailed results that are generated. You configure the tool to specify one or more target platforms, such as .NET Core 3.1 or .NET Standard 2.0. Results are provided with details for each platform targeted. Figure 3-4 shows an example of the tool's output.

	A	B	C	D	E	F
1	Target type	Target member	Assembly Name	.NET Core + Platform Extensions,Version=	.NET Core,Version=v3.1	.NET Standard + Platform Exte
86	T:System.Web.HttpRequestBase	T:System.Web.HttpRequestBase	eShopLegacyMVC	Not supported	Not supported	Not supported
125	T:System.Web.HttpRequestBase	M:System.Web.HttpRequestBase.get_Url	eShopLegacyMVC	Not supported	Not supported	Not supported
363	T:System.Web.SessionState.HttpSessionState	T:System.Web.SessionState.HttpSessionState	eShopLegacyMVC	Not supported	Not supported	Not supported
369	T:System.Web.SessionState.HttpSessionState	M:System.Web.SessionState.HttpSessionState.set_	eShopLegacyMVC	Not supported	Not supported	Not supported
405	T:System.Web.HttpApplication	T:System.Web.HttpApplication	eShopLegacyMVC	Not supported	Not supported	Not supported
410	T:System.Web.HttpApplication	M:System.Web.HttpApplication.#ctor	eShopLegacyMVC	Not supported	Not supported	Not supported
567	T:System.Web.Routing.RouteCollection	T:System.Web.Routing.RouteCollection	eShopLegacyMVC	Not supported	Not supported	Not supported
616	T:System.Web.HttpRequest	T:System.Web.HttpRequest	eShopLegacyMVC	Not supported	Not supported	Not supported
624	T:System.Web.HttpRequest	M:System.Web.HttpRequest.get_RawUrl	eShopLegacyMVC	Not supported	Not supported	Not supported
629	T:System.Web.HttpRequest	M:System.Web.HttpRequest.get_UserAgent	eShopLegacyMVC	Not supported	Not supported	Not supported
672	T:System.Web.Routing.Route	T:System.Web.Routing.Route	eShopLegacyMVC	Not supported	Not supported	Not supported
775	T:System.Web.HttpContext	T:System.Web.HttpContext	eShopLegacyMVC	Not supported	Not supported	Not supported
783	T:System.Web.HttpContext	M:System.Web.HttpContext.get_Current	eShopLegacyMVC	Not supported	Not supported	Not supported
802	T:System.Web.HttpContext	M:System.Web.HttpContext.get_Request	eShopLegacyMVC	Not supported	Not supported	Not supported
815	T:System.Web.HttpContext	M:System.Web.HttpContext.get_Session	eShopLegacyMVC	Not supported	Not supported	Not supported
1124	T:System.Web.HttpServerUtilityBase	T:System.Web.HttpServerUtilityBase	eShopLegacyMVC	Not supported	Not supported	Not supported
1128	T:System.Web.HttpServerUtilityBase	M:System.Web.HttpServerUtilityBase.MapPath(Sy	eShopLegacyMVC	Not supported	Not supported	Not supported
1165	T:System.Web.Hosting.HostingEnvironment	T:System.Web.Hosting.HostingEnvironment	eShopLegacyMVC	Not supported	Not supported	Not supported
1169	T:System.Web.Hosting.HostingEnvironment	M:System.Web.Hosting.HostingEnvironment.get_A	eShopLegacyMVC	Not supported	Not supported	Not supported
1322	T:System.Web.Routing.RouteTable	T:System.Web.Routing.RouteTable	eShopLegacyMVC	Not supported	Not supported	Not supported
1324	T:System.Web.Routing.RouteTable	M:System.Web.Routing.RouteTable.get_Routes	eShopLegacyMVC	Not supported	Not supported	Not supported
1365						
1366						
1367						
1368						

Portability Summary | **Details** | Missing assemblies | ⊕

Figure 3-4. .NET Portability Analyzer report details.

Update class library dependencies

Large apps typically involve multiple projects, and most projects other than the ASP.NET MVC web project are likely to be class libraries. Class libraries tend to be the easiest to port between .NET

platforms, especially compared to ASP.NET projects, which are among the most difficult (and typically need to be re-created).

Teams can consider the try-convert tool for migrating class libraries to .NET Core. The try-convert tool analyzes a .NET Framework project file and attempts to migrate it to the .NET Core project file format, making any modifications it can safely perform in the process. This tool won't work on an ASP.NET project, but can help speed up the process of migrating class libraries.

The try-convert tool is deployed as a .NET Core command line tool. It only runs on Windows, since it's designed to work with .NET Framework apps. You can install it by running the following command from a command prompt:

```
dotnet tool install -g try-convert
```

Once you've successfully installed the tool, you can run try-convert in the folder where the class library's project file is located.

Update NuGet package dependencies

Analyze your use of third-party NuGet packages and determine if any of them don't yet support .NET Standard (or do support it but only with a new version). It can be helpful to update NuGet packages to use <PackageReference> syntax using Visual Studio's converter tool, so that top-level dependencies are visible. Next, check whether the current or later versions of these packages support .NET Core or .NET Standard. This information can be found on [nuget.org] or within Visual Studio for each package.

If support exists using the version of the package the app currently uses, great! If not, see if a more recent version of the package has the support and research what would be involved in upgrading. There may be breaking changes in the package, especially if the major version of the package changes between your currently used version and the one to which you're upgrading.

In some cases, no version of a given package works with .NET Core. In that case, teams have a couple options. They can continue depending on the .NET Framework version, but this has limitations. The app will only run on Windows, and the team may want to run Portability Analyzer on the package's binaries to see if there are any issues likely to be encountered. Certainly the team will want to test thoroughly. The other option is to find a different package or, if the required package is open source, upgrade it to .NET Standard or .NET Core themselves.

Migrate ASP.NET MVC projects

The System.Web namespace and types don't exist in .NET Core. When you're analyzing dependencies and using tools like try-convert, you'll find they don't offer many suggestions for automatic migration of ASP.NET MVC projects and any code in them that references System.Web. For these projects, you'll need to start with a new ASP.NET Core web project and manually migrate files to this project.

In general, it's a good practice to minimize how much of an app's business logic lives in its user interface layer. It's also best to keep controllers and views small. Apps that have followed this guidance will be easier to port than those that have a significant amount of their logic in the ASP.NET web project. If you have an app you're considering porting, but haven't begun the process yet, keep

this in mind as you maintain it. Any effort you put toward minimizing how much code is in the ASP.NET MVC or Web API project will likely result in less work when the time comes to port the app.

The next chapter digs into details of how to migrate from ASP.NET MVC and Web API projects to ASP.NET Core projects. The previous chapter called out the biggest differences between the apps. Once the basic project structure is in place, migrating individual controllers and views is usually straightforward, especially if they're mainly focused on web responsibilities.

References

- [try-convert tool](#)
- [apiport tool](#)

Strategies for migrating while running in production

Many teams have .NET Framework apps they plan to migrate to .NET Core, but the app is so large that the migration requires a significant amount of time to complete. The original app needs to live on while the migration is done piece by piece. There needs to be a way for the old and new versions of the app to work together side-by-side, or for the old version to be migrated in-place, at least some of the way, without breaking it. Teams can employ many different strategies to support these goals.

Refactor the .NET Framework solution

A good place to start if you plan to port a .NET Framework app to .NET Core is to refactor it to work better with .NET Core. This means updating individual class libraries to target .NET Standard and moving as much logic out of your ASP.NET MVC projects and into these class libraries. Any code you have in .NET Standard libraries should move relatively easily from .NET Framework to .NET Core.

When refactoring, make sure you're following good refactoring fundamentals. For example, create tests that verify what the system does before you start refactoring. Run these tests when you're done to confirm you didn't change the system's behavior. You may need to add characterization tests to the system if you don't already have a good suite of automated tests you can rely on.

Extract front-end assets to a CDN

If your .NET Framework apps include a lot of static assets, like scripts, CSS files, or images, you may be able to migrate these to a separate CDN. Then, update the existing app to reference the CDN links for these assets. When you port the app to .NET Core, these static files won't be part of the migration, and you'll just continue referencing them from the CDN in the ASP.NET Core app.

Extract and migrate individual microservices

Large .NET Framework apps may already be comprised of separate front-end systems that can be migrated individually. Or they may be candidates for migration to a microservices architecture, with some pieces of existing ASP.NET MVC apps being pulled out into new ASP.NET Core microservice implementations. You can learn more about microservices in the associated ebook, [.NET Microservices: Architecture for Containerized .NET Applications](#).

For example, the existing app might have a set of features it uses related to user sign-in and registration. These could be migrated to a separate microservice, which could be built and deployed using ASP.NET Core and then integrated into the legacy .NET Framework app. Next, the app might have a few pages dedicated to tracking the individual user's shopping cart. These pages could also be pulled out into their own separate microservice and again integrated into the existing app. In this way, the original .NET Framework app continues running in production, but with more and more of its features coming from modernized .NET Core microservices.

Deploy multiple versions of the app side-by-side in IIS

Using a combination of host headers and redirects, an existing ASP.NET MVC app can be configured to run side by side with an ASP.NET Core app on the same IIS server. As pieces of functionality, such as individual controllers, are ported to ASP.NET Core, their routes and URLs are mapped within IIS to target the ASP.NET Core web site or sub-application (IIS virtual directories aren't supported with ASP.NET Core apps). An ASP.NET Core app can be hosted as an IIS sub-application (sub-app). The sub-app's path becomes part of the root app's URL.

Apply the Strangler pattern

Large ASP.NET MVC apps can be gradually replaced with a new ASP.NET Core app by incrementally migrating pieces of functionality. One approach to this is called the strangler pattern, named for strangler vines that strangle and eventually tear down trees. This approach relies on first implementing a facade layer over top of the existing solution. This facade should be built using the new approach to the problem, or an off-the-shelf solution such as an API gateway.

Once the facade is in place, you can route part of it to a new ASP.NET Core app. As you port more of the original .NET Framework app to .NET Core, you continue to update the facade layer accordingly, sending more of the facade's total functionality to the new system. Figure 3-5 shows the strangler pattern progression over time.

Multi-targeting approaches

Large apps that target .NET Framework may be migrated to ASP.NET Core over time by using multi-targeting and separate code paths for each framework. For example, code that must run in both environments could be modified with preprocessor #if directives to implement different functionality or use different dependencies when run in .NET Framework versus .NET Core. Another option is to modify project files to include different sets of files based on which framework is being targeted. Project files can use different globbing patterns, such as *.core.cs, to include different sets of source files depending on the framework being targeted.

These techniques allow a single common codebase to be maintained while new functionality is added and (parts of) the app are ported to use .NET Core.
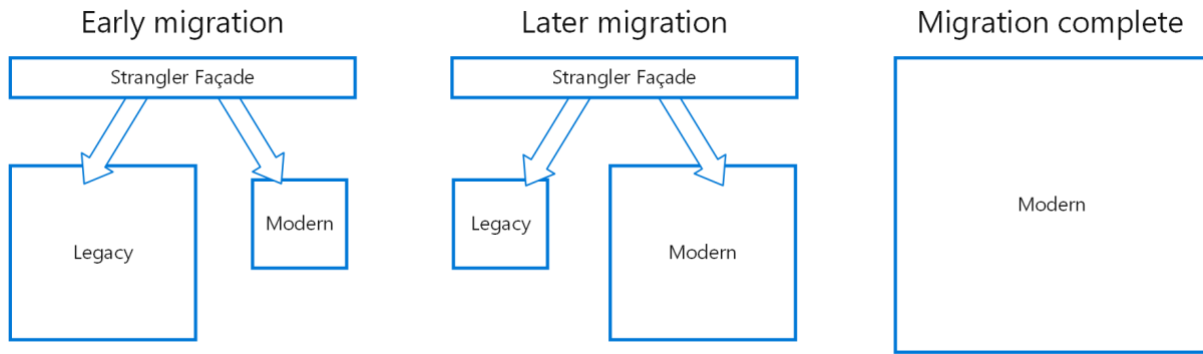
Figure 3-5. The Strangler pattern over time.

Eventually, the entire facade layer corresponds to the new, modern implementation. At this point, both the legacy system and the face layer can be retired.

Summary

Frequently, large ASP.NET MVC and Web API apps won't be ported to ASP.NET Core all at once, but will migrate incrementally over time. This section offers several strategies for performing this incremental migration. Choose the one(s) that will work best for your organization and app.

References

- [.NET Microservices: Architecture for Containerized .NET Applications](#)
- [eShopOnContainers Reference Microservices Application](#)
- [Host ASP.NET Core on Windows with IIS](#)
- [Strangler pattern](#)

Example migration of eShop to ASP.NET Core

In this chapter, you'll see how to migrate a .NET Framework app to .NET Core. The chapter examines a sample online store app written for ASP.NET 5.0. The app will use many of the concepts and tools described earlier in this book. You'll find the starting point app in the *eShopModernizing* GitHub repository. There are several different starting point apps. This chapter focuses on the *eShopLegacyMVCSolution*.

The initial version of the project is shown in Figure 4-1. It's a fairly standard ASP.NET MVC 5 app.
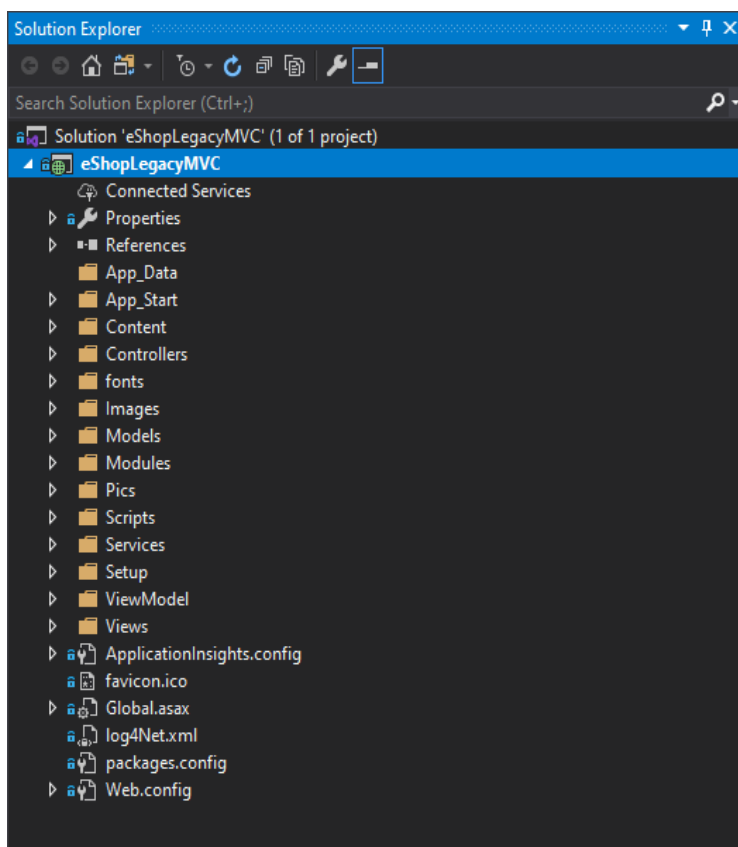
Figure 4-1. The eShopModernizing MVC sample project structure.

Run *ApiPort* to identify problematic APIs

The first step in preparing to migrate is to run the *ApiPort* tool. The tool identifies how many .NET Framework APIs the app calls and how many of these have .NET Standard or .NET Core equivalents. Focus primarily on your own app's logic, not third-party dependencies, and pay attention to System.Web dependencies that will need to be ported. The ApiPort tool was introduced in the last chapter on understanding and updating dependencies.

After installing and configuring the *ApiPort* tool, run the analysis from within Visual Studio, as shown in Figure 4-2.
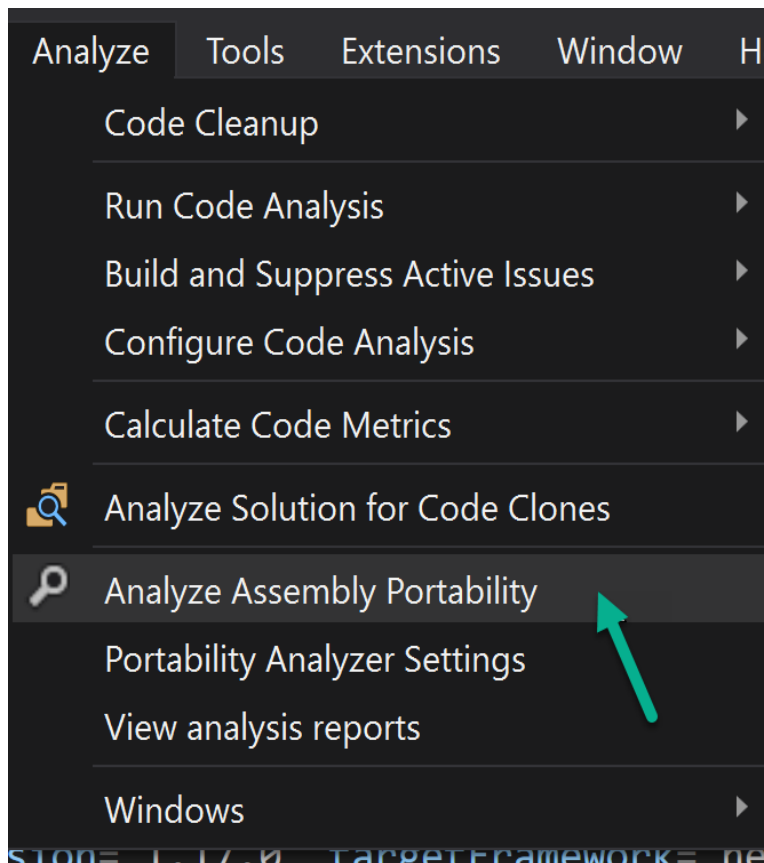
Figure 4-2. Analyze assembly portability in Visual Studio.

Choose the web project's assembly from the project's *bin* folder, as shown in Figure 4-3.
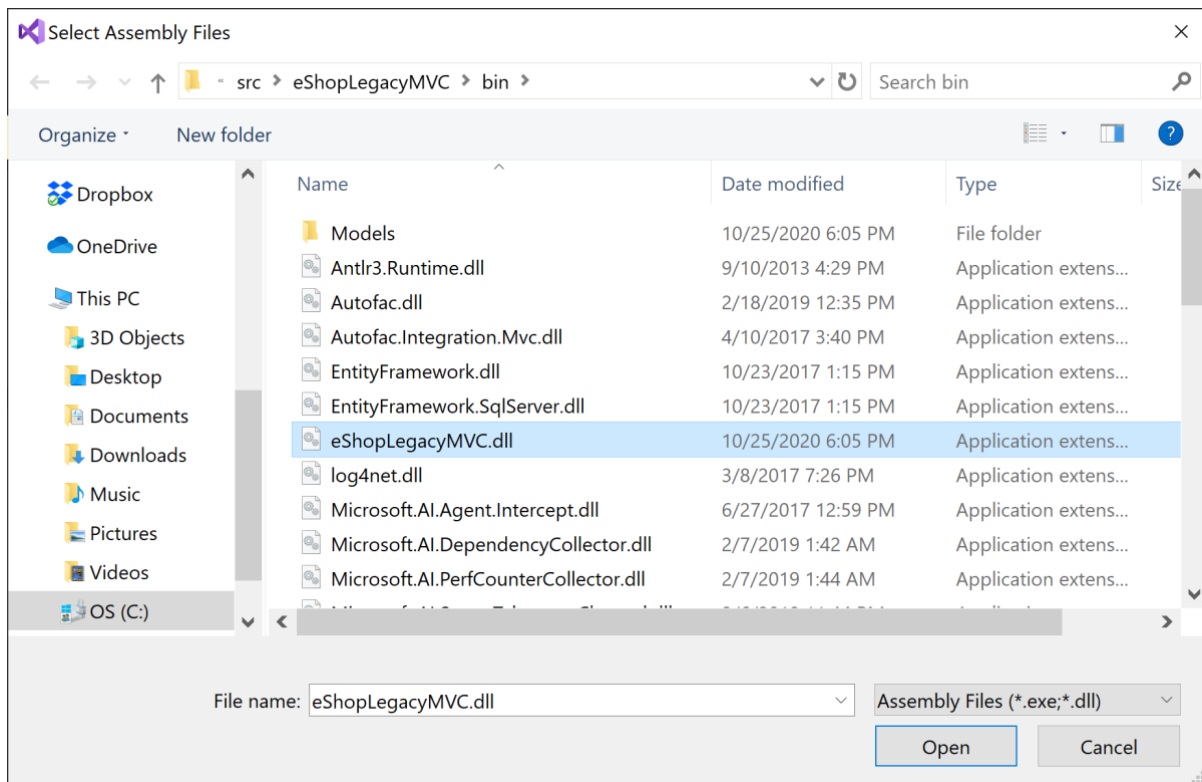
Figure 4-3. Choose the project's web assembly.

If your solution includes several projects, you can choose all of them. The *eShop* sample includes just a single MVC project.

Once the report is generated, open the file and review the results. The summary provides a high-level view of what percentage of .NET Framework calls your app is making have compatible versions. Figure 4-4 shows the summary for the *eShop* MVC project.

Figure 4-4. ApiPort summary.

For this app, about 80 percent of the .NET Framework calls are compatible. 20 percent of the calls need to be addressed during the porting process. Viewing the details reveals that all of the incompatible calls are part of System.Web, which is an expected incompatibility. The dependencies on System.Web calls will be addressed when the app's controllers and related classes are migrated in a later step. Figure 4-5 lists some of the specific types found by the tool:

Target type	Target member	Assembly Name
T:System.Web.Mvc.UrlParameter	T:System.Web.Mvc.UrlParameter	eShopLegacyMVC
T:System.Web.Mvc.UrlParameter	F:System.Web.Mvc.UrlParameter.Optional	eShopLegacyMVC
T:System.Web.HttpApplication	T:System.Web.HttpApplication	eShopLegacyMVC
T:System.Web.HttpApplication	M:System.Web.HttpApplication.#ctor	eShopLegacyMVC
T:System.Web.Mvc.HttpNotFoundResult	T:System.Web.Mvc.HttpNotFoundResult	eShopLegacyMVC
T:System.Web.Mvc.Controller	T:System.Web.Mvc.Controller	eShopLegacyMVC
T:System.Web.Mvc.Controller	M:System.Web.Mvc.Controller.#ctor	eShopLegacyMVC
T:System.Web.Mvc.Controller	M:System.Web.Mvc.Controller.Dispose(System.Bo	eShopLegacyMVC
T:System.Web.Mvc.Controller	M:System.Web.Mvc.Controller.File(System.Byte[],	eShopLegacyMVC
T:System.Web.Mvc.Controller	M:System.Web.Mvc.Controller.get_ModelState	eShopLegacyMVC
T:System.Web.Mvc.Controller	M:System.Web.Mvc.Controller.get_Request	eShopLegacyMVC
T:System.Web.Mvc.Controller	M:System.Web.Mvc.Controller.get_Server	eShopLegacyMVC
T:System.Web.Mvc.Controller	M:System.Web.Mvc.Controller.get_Url	eShopLegacyMVC
T:System.Web.Mvc.Controller	M:System.Web.Mvc.Controller.HttpNotFound	eShopLegacyMVC
T:System.Web.Mvc.Controller	M:System.Web.Mvc.Controller.RedirectToAction(S	eShopLegacyMVC
T:System.Web.Mvc.Controller	M:System.Web.Mvc.Controller.View(System.Objec	eShopLegacyMVC
T:System.Web.Mvc.ActionNameAttribute	T:System.Web.Mvc.ActionNameAttribute	eShopLegacyMVC
T:System.Web.Mvc.ActionNameAttribute	M:System.Web.Mvc.ActionNameAttribute.#ctor(S	eShopLegacyMVC
T:System.Web.HttpContext	T:System.Web.HttpContext	eShopLegacyMVC
T:System.Web.HttpContext	M:System.Web.HttpContext.get_Current	eShopLegacyMVC
T:System.Web.HttpContext	M:System.Web.HttpContext.get_Request	eShopLegacyMVC
T:System.Web.HttpContext	M:System.Web.HttpContext.get_Session	eShopLegacyMVC
T:System.Web.Mvc.HandleErrorAttribute	T:System.Web.Mvc.HandleErrorAttribute	eShopLegacyMVC
T:System.Web.Mvc.HandleErrorAttribute	M:System.Web.Mvc.HandleErrorAttribute.#ctor	eShopLegacyMVC
T:System.Web.Mvc.HttpGetAttribute	T:System.Web.Mvc.HttpGetAttribute	eShopLegacyMVC
T:System.Web.Mvc.HttpGetAttribute	M:System.Web.Mvc.HttpGetAttribute.#ctor	eShopLegacyMVC

Figure 4-5. ApiPort incompatible type details.

Most of the incompatible types refer to Controller and various related attributes that have equivalents in ASP.NET Core.

Update project files and NuGet reference syntax

Next, migrate from the older *.csproj* file structure to the newer, simpler structure introduced with .NET Core. In doing so, you'll also migrate from using a *packages.config* file for NuGet references to using <PackageReference> elements in the project file.

The original project's *eShopLegacyMVC.csproj* file is 418 lines long. A sample of the project file is shown in Figure 4-6. To offer a sense of its overall size and complexity, the right side of the image contains a miniature view of the entire file.

Figure 4-6. The eShopLegacyMVC.csproj file structure.

A common way to create a new project file for an existing ASP.NET project is to create a new ASP.NET Core app using dotnet new or **File** > **New** > **Project** in Visual Studio. Then files can be copied from the old project to the new one to complete the migration.

In addition to the C# project file, NuGet dependencies are stored in a separate 45-line *packages.config* file, as shown in Figure 4-7.

Figure 4-7. The packages.config file.

After upgrading to the new *.csproj* file format, you can migrate *packages.config* in class library projects using Visual Studio. This functionality doesn't work with ASP.NET projects, however. Learn more about migrating *packages.config* to <PackageReference> in Visual Studio. If you have a large number of projects to migrate, this community tool may help.

Create new ASP.NET Core project

Add a new ASP.NET Core project to the existing app's solution to make moving files easier, as most of the work can be done from within Visual Studio's **Solution Explorer**. In Visual Studio, right-click on your app's solution and choose **Add New Project**. Choose **ASP.NET Core web application**, and give the new project a name as shown in Figure 4-8.
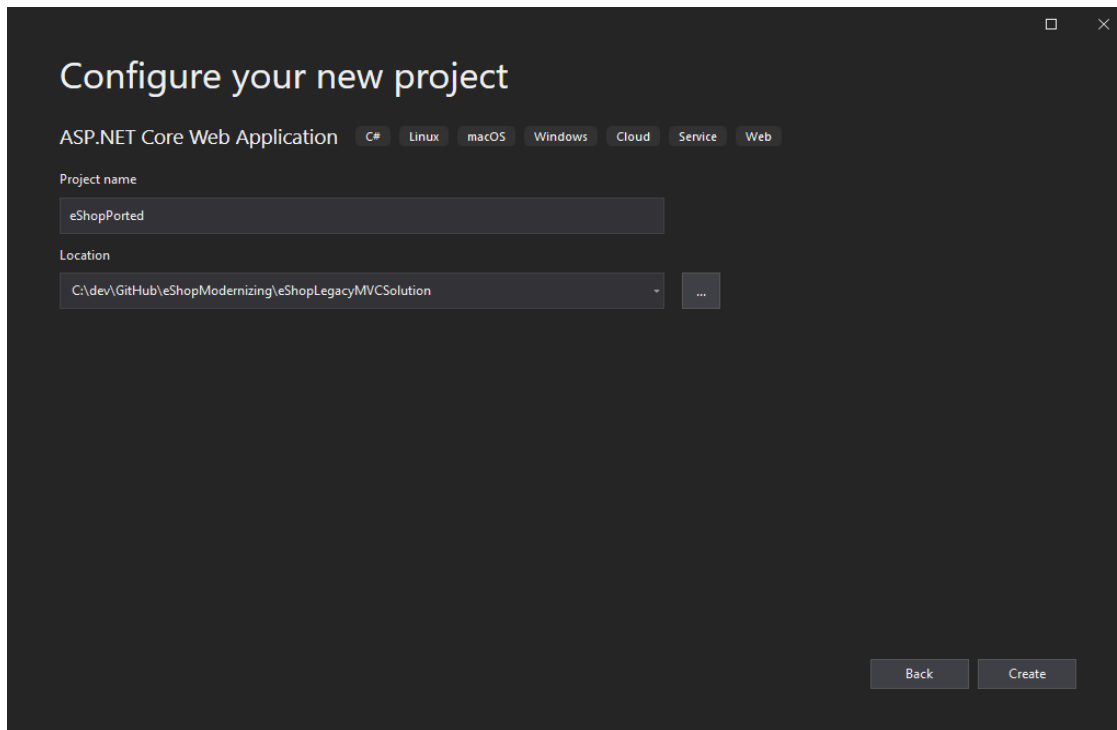
Figure 4-8. Add new ASP.NET Core web application.

The next dialog will ask you to choose which template to use. Select the **Empty** template. Be sure to also change the dropdown from **.NET Core** to **.NET Framework**. Select **ASP.NET Core 2.2**, as shown in Figure 4-9.
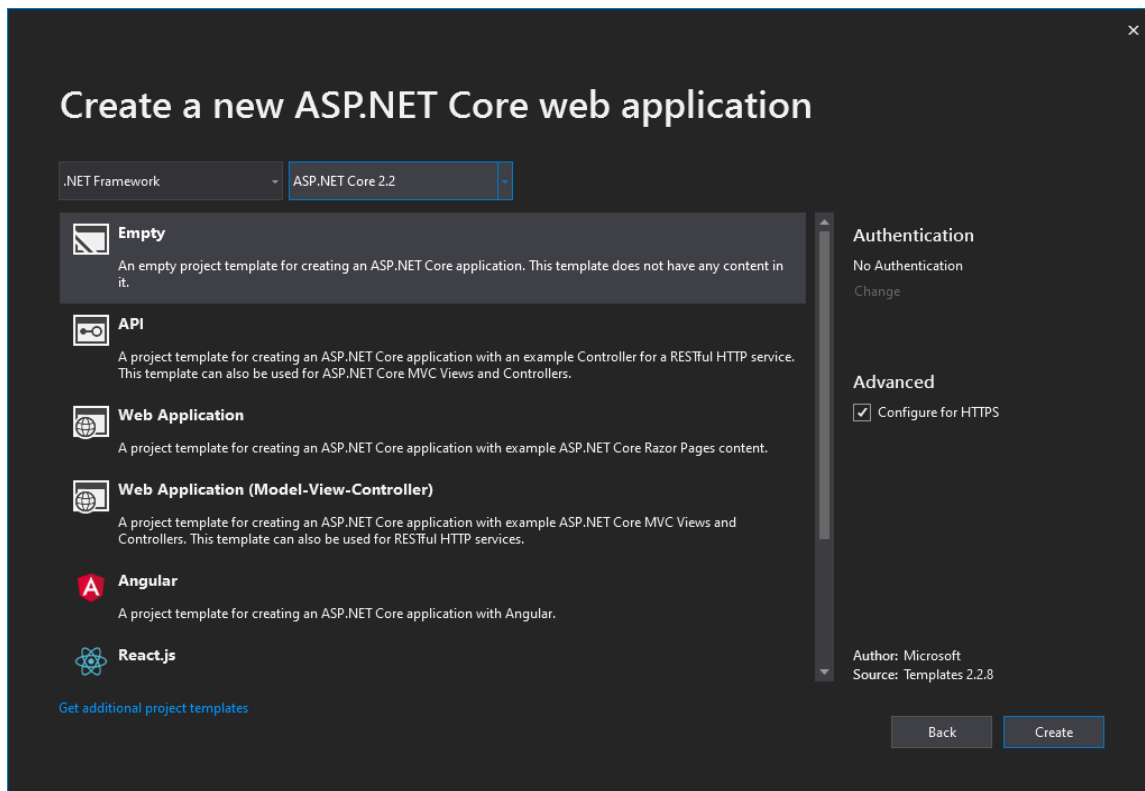
Figure 4-9. Choose an Empty project template targeting .NET Framework with ASP.NET Core 2.2.

Migrating NuGet Packages

Since the built-in migration tool for migrating *packages.config* to <PackageReference> doesn't work on ASP.NET projects, you can use a community tool instead, or migrate by hand. A community tool I've used uses an XSL file to transform from one format to the other. To use it, first copy the *packages.config* file to the newly created ASP.NET Core project folder. Make a backup of your files, as this script removes the *packages.config* file from all folders under where you run the script. Then run these commands from the project folder (or for the entire solution if you prefer):

```
iwr https://git.io/vdKaV -OutFile Convert-ToPackageReference.ps1
iwr https://git.io/vdKar -OutFile  Convert-ToPackageReference.xsl
./Convert-ToPackageReference.ps1 | Out-Null
```

The first two commands download files so that they exist locally. The last line runs the script. After running it, try to build the new project. You'll most likely get some errors. To resolve them, you'll want to eliminate some references (like most of the Microsoft.AspNet and System packages), and you may need to remove some xmlns attributes.

In most ASP.NET MVC apps, many client-side dependencies like Bootstrap and jQuery were deployed using NuGet packages. In ASP.NET Core, NuGet packages are only used for server-side functionality. Client files should be managed through other means. Review the list of <PackageReference> elements added and remove and make note of any that are for client libraries, including:

- Bootstrap

- jQuery
- jQuery.Validation
- Modernizr
- popper.js
- Respond

The static client files installed by NuGet for these packages will be copied over to the new project's *wwwroot* folder and hosted from there. It's worth considering whether these files are still needed by the app, and whether it makes sense to continue hosting them or to use a content delivery network (CDN) instead. These library versions can be managed at build time using tools like [LibMan](#) or [npm](#). Figure 4-10 shows the full *eShopPorted.csproj* file after migrating package references using the conversion tool shown and removing unnecessary packages.

```
eShopPorted  >  ⤵ eShopPorted.csproj
 1    <Project Sdk="Microsoft.NET.Sdk.Web">
 2      <PropertyGroup>
 3        <TargetFramework>net461</TargetFramework>
 4      </PropertyGroup>
 5      <ItemGroup>
 6        <PackageReference Include="Microsoft.AspNetCore" Version="2.2.0" />
 7        <PackageReference Include="Antlr">
 8          <Version>3.5.0.2</Version>
 9        </PackageReference>
10        <PackageReference Include="Autofac">
11          <Version>4.9.1</Version>
12        </PackageReference>
13        <PackageReference Include="Autofac.Mvc5">
14          <Version>4.0.2</Version>
15        </PackageReference>
16        <PackageReference Include="EntityFramework">
17          <Version>6.2.0</Version>
18        </PackageReference>
19        <PackageReference Include="log4net">
20          <Version>2.0.8</Version>
21        </PackageReference>
22        <PackageReference Include="Newtonsoft.Json">
23          <Version>12.0.1</Version>
24        </PackageReference>
25        <PackageReference Include="WebGrease">
26          <Version>1.6.0</Version>
27        </PackageReference>
28      </ItemGroup>
29    </Project>
```

Figure 4-10. Package references in the eShopPorted.csproj file.

The package references can be further compacted by making the <Version>1.0.0.0</Version> element a Version=1.0.0.0 attribute on <PackageReference>.

Migrate static files

Any static files the app uses, including third-party scripts and frameworks but also custom images and stylesheets, must be copied from the old project to the new one. In ASP.NET MVC apps, files were typically accessed based on their location within the project folder. In ASP.NET Core apps, these static files will be accessed based on their location within the *wwwroot* folder. For the *eShop* project, there are static files in the following folders:

- *Content*
- *fonts*
- *Images*
- *Pics*
- *Scripts*

The **Empty** project template used in the previous step doesn't include this folder by default, or the middleware needed for it to work. You'll need to add them.

Add a *wwwroot* folder to the root of the project.

Add version 2.2.0 of the Microsoft.AspNetCore.StaticFiles NuGet package.

In *Startup.cs*, add a call to app.UseStaticFiles() in the Configure method:

```csharp
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseStaticFiles();

    // ...
}
```

Copy the *Content* folder from the ASP.NET MVC app to the new project's *wwwroot* folder.

Run the app and navigate to its */Content/base.css* folder to verify that the static file is served correctly from its expected path. Continue copying the rest of the folders containing static files to the new project. You'll also want to copy the *favicon.ico* file from the project's root to the *wwwroot* folder. Figure 4-11 shows the results after these files and their folders have all been copied.
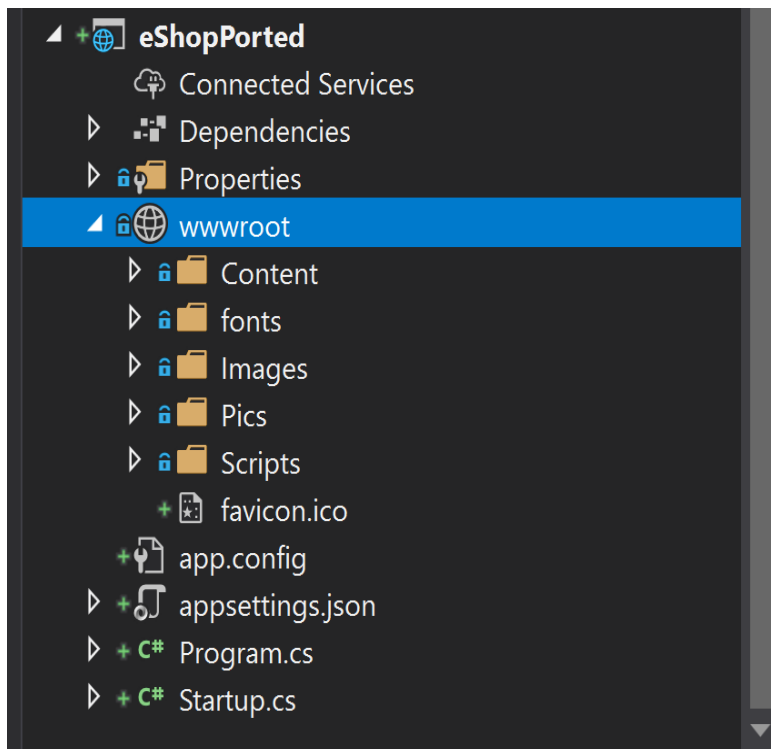
Figure 4-11. Static folders copied over to wwwroot folder.

Migrate C# files

Next, copy over the C# files used by the app, including standard MVC folders and their contents like *Controllers*, *Models*, *ViewModel*, and *Services*. There will most likely be some changes needed in these files. It's best to copy one folder (or subfolder) at a time and compile to see what errors need to be addressed as you go.

For the *eShop* sample, the first folder I choose to migrate is the *Models* folder, which includes C# entities and Entity Framework classes. This folder's classes are used by most of the others, so they won't work until these classes have been copied. After copying the folder and building, the compiler revealed errors related to missing namespace System.Web.Hosting, related access to HostingEnvironment, and a reference to ConfigurationManager.AppSettings. The solution to these issues will be to pass in the necessary path data; for now the breaking lines are commented out and a TODO: comment is added to each one to track it. After changing five lines, the **Task List** shows five items and the project builds.

Next, the *ViewModel* folder, with its one class, is copied over. It's an easy one, and builds immediately.

The *Services* folder is copied over. This folder's classes depend on Entity Framework classes from the *Models* folder, which is why it needed to be copied after that folder. Fortunately, it too builds without errors.

That leaves the *Controllers* folder and its two Controller classes. After copying the folder to the new project and building, there are seven build errors. Four of them are related to ViewBag access and report an error of:

Missing compiler required member 'Microsoft.CSharp.RuntimeBinder.CSharpArgumentInfo.Create'

To resolve this error add a NuGet package reference to C#:

```
<PackageReference Include="Microsoft.CSharp" Version="4.7.0" />
```

The remaining three errors specify types that are defined in an assembly that isn't referenced. Specifically these types:

- HttpServerUtilityBase
- RouteValueDictionary
- HttpRequestBase

Let's look at each error one by one. The first error occurs while trying to reference the Server property of Controller, which no longer exists. The goal of the operation is to get the path to an image file in the app:

```
if (item != null)
{
    var webRoot = Server.MapPath("~/Pics"); // compiler error on this line
    var path = Path.Combine(webRoot, item.PictureFileName);

    string imageFileExtension = Path.GetExtension(item.PictureFileName);
    string mimetype = GetImageMimeTypeFromImageFileExtension(imageFileExtension);

    var buffer = System.IO.File.ReadAllBytes(path);

    return File(buffer, mimetype);
}
```

There are two possible solutions to this problem. The first is to keep the functionality as it is. In this case, rather than using Server.MapPath, a fixed path referencing the image files' location in *wwwroot* should be used. Alternately, since the only purpose of this action method is to return a static image file, the references to this action in view files can be updated to reference the static files directly, which improves runtime performance. Since no processing is being done as part of this action, there's no reason not to just serve the files directly. If it's not tenable to update all references to this action, the action could be rewritten to produce a redirect to the static file's location.

The next two errors both occur in the same private method in the same line of code:

```
private void AddUriPlaceHolder(CatalogItem item)
{
    item.PictureUri = this.Url.RouteUrl(PicController.GetPicRouteName, new { catalogItemId
= item.Id }, this.Request.Url.Scheme);
}
```

Both this.Url and this.Request cause compiler errors. Looking at how this code is used, its purpose is to build a link to the PicController action that renders image files. The same one we just discovered could probably be replaced with direct links to the static files located in *wwwroot*. For now, it's worth commenting out this code and adding a TODO: comment to reference the pics another way.

It's worth noting that the base Controller class, used by the CatalogController class in which this code appears, is still referring to System.Web.Mvc.Controller. There will undoubtedly be more errors to fix once we update this to use ASP.NET Core. First, remove the using System.Web.Mvc; line from the list

of using statements in CatalogController. Next, add the NuGet package Microsoft.AspNetCore.Mvc. Finally, add a using Microsoft.AspNetCore.Mvc; statement, and build the app again.

This time, there are 16 errors:

- Include is not a valid named attribute argument (2)
- HttpStatusCodeResult not found (3)
- HttpNotFound does not exist (3)
- SelectList not found (8)

Once more, let's review these errors one by one. First, SelectList can be fixed by adding using Microsoft.AspNetCore.Mvc.Rendering;, which eliminates half of the errors.

All references to return HttpNotFound(); should be replaced with return NotFound();.

All references to return new HttpStatusCodeResult(HttpStatusCode.BadRequest); should be replaced with return BadRequest();.

That just leaves the use of Include with a [Bind] attribute on a couple of action methods that look like this:

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create([Bind(Include =
"Id,Name,Description,Price,PictureFileName,CatalogTypeId,CatalogBrandId,AvailableStock,Rest
ockThreshold,MaxStockThreshold,OnReorder")] CatalogItem catalogItem)
{
```

The preceding code restricts model binding to the properties listed in the Include string. In ASP.NET Core MVC, the [Bind] attribute still exists, but no longer needs the Include = argument. Pass the list of properties directly to the [Bind] attribute:

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult
Create([Bind("Id,Name,Description,Price,PictureFileName,CatalogTypeId,CatalogBrandId,Availa
bleStock,RestockThreshold,MaxStockThreshold,OnReorder")] CatalogItem catalogItem)
{
```

With these changes, the project compiles once more. It's generally a better practice to use separate model types for controller inputs, rather than using model binding directly to your domain model or data model types.

Migrate views

The two biggest ASP.NET Core MVC features related to views are Razor Pages and Tag Helpers. For the initial migration, we won't use either feature. You should, however, keep the features in mind if you continue supporting the app once it's been migrated. The next step is to copy the *Views* folder from the original project into the new one. After building, there are nine errors:

- HttpContext does not exist (2)
- Scripts does not exist (5)

- Styles does not exist (1)
- HtmlString could not be found(1)

Investigating these errors finds that most of them are in the main *_Layout.cshtml*, *with several related to rendering script and style tags, or displaying when the server hosting the app was last restarted. The following code listing shows problem areas in the _Layout.cshtml* file:*

```
// other lines omitted; only errors shown
@Styles.Render("~/Content/css")
@Scripts.Render("~/bundles/modernizr")

@{ var sessionInfo = new HtmlString($"{HttpContext.Current.Session["MachineName"]},
{HttpContext.Current.Session["SessionStartTime"]}");}

@Scripts.Render("~/bundles/jquery")
@Scripts.Render("~/bundles/bootstrap")
```

The reference to Modernizr can be removed. The references to Bootstrap and jQuery can be replaced with CDN links to the appropriate version.

Replace @Styles.Render line with:

```
<link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css"
integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
crossorigin="anonymous">
```

Replace the last two Scripts.Render lines with:

```
<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js" integrity="sha384-
q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo"
crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.js"
integrity="sha384-UO2eT0CpHqdSJQ6hJty5KVphtPhzWj9WO1clHTMGa3JDZwrnQq4sF86dIHNDz0W1"
crossorigin="anonymous"></script>
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js"
integrity="sha384-JjSmVgyd0p3pXB1rRibZUAYoIIy6OrQ6VrjIEaFf/nJGzIxFDsf4x0xIM+B07jRM"
crossorigin="anonymous"></script>
```

Finally, after the Bootstrap <link>, add additional <link> elements for local styles your app uses. For *eShop*, the result is shown here:

```
<link rel="stylesheet" href="~/Content/custom.css" />
<link rel="stylesheet" href="~/Content/base.css" />
<link rel="stylesheet" href="~/Content/Site.css" />
```

To determine the order in which the <link> elements should appear, look at your original app's rendered HTML. Alternatively, review *BundleConfig.cs*, which for the *eShop* sample includes this code indicating the appropriate sequence:

```
bundles.Add(new StyleBundle("~/Content/css").Include(
        "~/Content/bootstrap.css",
        "~/Content/custom.css",
        "~/Content/base.css",
        "~/Content/site.css"));
```

Building again reveals one more error loading jQuery Validation on the *Create* and *Edit* views. Replace it with this script:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery-
validate/1.17.0/jquery.validate.min.js" integrity="sha512-
O/nUTF5mdFkhEoQHFn9N5wmgYyW323JO6v8kr6ltSRKriZyTr/8417taVWeabVS4iONGk2V444QD0P2cwhuTkg=="
crossorigin="anonymous"></script>
```

The last thing to fix in the views is the reference to Session to display how long the app has been running, and on which machine. We can collect this data in Startup as static variables and display the variables on the layout page. Add the following properties to *Startup.cs*:

```
public static DateTime StartTime { get; } = DateTime.UtcNow;
public static string MachineName { get; } = Environment.MachineName;
```

Then replace the content of the footer in the layout with the following code:

```
<section class="col-sm-6">
    <img class="esh-app-footer-text hidden-xs" src="~/images/main_footer_text.png"
width="335" height="26" alt="footer text image" />
    <br />
<small>@eShopPorted.Startup.MachineName - @eShopPorted.Startup.StartTime.ToString()
UTC</small>
</section>
```

At this point, the app once more builds successfully. However, trying to run it just yields *Hello World!* because the **Empty** ASP.NET Core template is only configured to display that in response to any request. In the next section, I complete the migration by configuring the app to use ASP.NET Core MVC, including dependency injection and configuration. Once that's in place, the app should run. Then it will be time to fix the TODO: tasks that were created earlier.

Migrate app startup components

The last migration step is to take the app startup tasks from *Global.asax*, and the classes it calls, and migrate these to their ASP.NET Core equivalents. These tasks include configuration of MVC itself, setting up dependency injection, and working with the new configuration system. In ASP.NET Core, these tasks are handled in the *Startup.cs* file.

Configure MVC

The original ASP.NET MVC app has the following code in its Application_Start in *Global.asax*, which runs when the app starts up:

```
protected void Application_Start()
{
    container = RegisterContainer();
    AreaRegistration.RegisterAllAreas();
    FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
    RouteConfig.RegisterRoutes(RouteTable.Routes);
    BundleConfig.RegisterBundles(BundleTable.Bundles);
    ConfigDataBase();
}
```

Looking at these lines one by one, the RegisterContainer method sets up dependency injection, which will be ported below. The next three lines configure different parts of MVC: areas, filters, and routes. Bundles are replaced by static files in the ported app. The last line sets up data access for the app, which will be shown in a later section.

Since this app isn't actually using areas, there's nothing that needs to be done to migrate the area registration call. If your app does need to migrate areas, the docs specify how to configure areas in ASP.NET Core.

The call to register global filters invokes a helper on the FilterConfig class in the app's *App_Start* folder:

```
public static void RegisterGlobalFilters(GlobalFilterCollection filters)
{
    filters.Add(new HandleErrorAttribute());
}
```

The only attribute added to the app is the ASP.NET MVC filter, HandleErrorAttribute. This filter ensures that when an exception occurs as part of a request, a default action and view are displayed, rather than the exception details. In ASP.NET Core, this same functionality is performed by the UseExceptionHandler middleware. The detailed error messages aren't enabled by default. They must be configured using the UseDeveloperExceptionPage middleware. To configure this behavior to match the original app, the following code must be added to the start of the Configure method in *Startup.cs*:

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

This takes care of the only filter used by the eShop app, and in this case it was done by using built-in middleware. If you have global filters that must be configured in your app, this is done when MVC is added in the ConfigureServices method, which is shown later in this chapter.

The last piece of MVC-related logic that needs to be migrated are the app's default routes. The call to RouteConfig.RegisterRoutes(RouteTable.Routes) passes the MVC route table to the RegisterRoutes helper method, where the following code is executed when the app starts up:

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.MapMvcAttributeRoutes();
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
        name: "Default",
        url: "{controller}/{action}/{id}",
        defaults: new { controller = "Catalog", action = "Index", id =
UrlParameter.Optional }
```

```
    );
}
```

Taking this code line-by-line, the first line sets up support for attribute routes. This is built into ASP.NET Core, so it's unnecessary to configure it separately. Likewise, *{resource}.axd* files aren't used with ASP.NET Core, so there's no need to ignore such routes. The MapRoute method configures the default for MVC, which uses the typical {controller}/{action}/{id} route template. It also specifies the defaults for this template, such that the CatalogController is the default controller used and the Index method is the default action. Larger apps will frequently include more calls to MapRoute to set up additional routes.

ASP.NET Core MVC supports [conventional routing and attribute routing](#). Conventional routing is analogous to how the route table is configured in the RegisterRoutes method listed previously. To set up conventional routing with a default route like the one used in the *eShop* app, add the following code to the bottom of the Configure method in *Startup.cs*:

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

> **Note**
>
> With ASP.NET Core 3.0 and later, this is changed to use endpoints. For the initial port to ASP.NET Core 2.2, this is the proper syntax for mapping conventional routes.

With these changes in place, the Configure method is almost done. The original template's app.Run method that prints *Hello World!* should be deleted. At this point, the method is as shown here:

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

    app.UseStaticFiles();

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

Now it's time to configure MVC services, followed by the rest of the app's support for dependency injection (DI). So far, the *eShopPorted* project's ConfigureServices method has remained empty. Now it's time to start populating it.

First, to get ASP.NET Core MVC to work properly, it needs to be added:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
}
```

The preceding code is the minimal configuration required to get MVC features working. There are many additional features that can be configured from this call, but for now this will suffice to build the app. Running it now routes the default request properly, but since we've not yet configured DI, an error occurs while activating CatalogController, because no implementation of type ICatalogService has been provided yet. We'll return to configure MVC further in a moment. For now, let's migrate the app's dependency injection.

Migrate dependency injection configuration

The original app's *Global.asax* file defines the following method, called when the app starts up:

```
protected IContainer RegisterContainer()
{
  var builder = new ContainerBuilder();

  builder.RegisterControllers(typeof(MvcApplication).Assembly);

  var mockData = bool.Parse(ConfigurationManager.AppSettings["UseMockData"]);
  builder.RegisterModule(new ApplicationModule(mockData));

  var container = builder.Build();
  DependencyResolver.SetResolver(new AutofacDependencyResolver(container));

  return container;
}
```

This code configures an Autofac container, reads a config setting to determine whether real or mock data should be used, and passes this setting into an Autofac module (found in the app's */Modules* directory). Fortunately, Autofac supports .NET Core, so the module can be migrated directly. Copy the folder into the new project and updates the class's namespace and it should compile.

ASP.NET Core has built-in support for dependency injection, but you can wire up a third-party container such as Autofac easily if necessary. In this case, since the app is already configured to use Autofac, the simplest solution is to maintain its usage. To do so, the ConfigureServices method signature must be modified to return an IServiceProvider, and the Autofac container instance must be configured and returned from the method.

Note: In .NET Core 3.0 and later, the process for integrating a third-party DI container has changed.

Part of configuring Autofac requires a call to builder.Populate(services). This extension is found in the Autofac.Extensions.DependencyInjection NuGet package, which must be installed before the code will compile.

After modifying ConfigureServices to configure an Autofac container, the new method is as shown here:

```
public IServiceProvider ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
```

```
    // Create Autofac container builder
    var builder = new ContainerBuilder();
    builder.Populate(services);
    bool useMockData = true; // TODO: read from config
    builder.RegisterModule(new ApplicationModule(useMockData));

    ILifetimeScope container = builder.Build();

    return new AutofacServiceProvider(container);
}
```

For now, the setting for useMockData is set to true. This setting will be read from configuration in a moment. At this point, the app compiles and should load successfully when run, as shown in Figure 4-12.

Figure 4-12. Ported eShop app running locally with mock data.

Migrate app settings

ASP.NET Core uses a new [configuration system](#), which by default leverages an *appsettings.json* file. By using CreateDefaultBuilder in *Program.cs*, the default configuration is already set up in the app. To access configuration, classes just need to request it in their constructor. The Startup class is no exception. To start accessing configuration in Startup and the rest of the app, request an instance of IConfiguration from its constructor:

```
public Startup(IConfiguration configuration)
{
    Configuration = configuration;
}

public IConfiguration Configuration { get; }
```

The original app referenced its settings using ConfigurationManager.AppSettings. A quick search for all references of this term yields the set of settings the new app needs. There are only two:

- UseMockData
- UseCustomizationData

If your app has more complex configuration, especially if it's using custom configuration sections, you'll probably want to create and bind objects to different parts of your app's configuration. These types can then be accessed using the options pattern. However, as noted in the referenced doc, this pattern shouldn't be used in ConfigureServices. Instead the ported app will reference the UseMockData configuration value directly.

First, modify the ported app's appsettings.json file and add the two settings in the root:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*",
  "UseMockData": "true",
  "UseCustomizationData" :   "true"
}
```

Now, modify ConfigureServices to access the UseMockData setting from the Configuration property (where previously we set the value to true):

```
    bool useMockData = Configuration.GetValue<bool>("UseMockData");
```

At this point, the setting is pulled from configuration. The other setting, UseCustomizationData, is used by the CatalogDBInitializer class. When you first ported this class, you commented out the access to ConfigurationManager.AppSettings["UseCustomizationData"]. Now it's time to modify it to use ASP.NET Core configuration. Modify the constructor of CatalogDBInitializer as follows:

```
    // add using Microsoft.Extensions.Configuration
    public CatalogDBInitializer(CatalogItemHiLoGenerator indexGenerator,
        IConfiguration configuration)
    {
        this.indexGenerator = indexGenerator;
        useCustomizationData = configuration.GetValue<bool>("UseCustomizationData");
    }
```

All access to configuration within the web app should be modified in this manner to use the new IConfiguration type. Dependencies that require access to .NET Framework configuration can include such settings in an *app.config* file added to the web project. The dependent projects can work with ConfigurationManager to access settings, and shouldn't require any changes if they already use this approach. However, since ASP.NET Core apps run as their own executable, they don't reference

web.config but rather *app.config*. By migrating settings from the legacy app's *web.config* file to a new *app.config* file in the ASP.NET Core app, components that use ConfigurationManager to access their settings will continue to function properly.

The app's migration is nearly complete. The only remaining task is data access configuration.

Data access considerations

ASP.NET Core apps running on .NET Framework can continue to leverage Entity Framework (EF). If performing an incremental migration, getting the app working with EF 6 before trying to port its data access to use EF Core may be worthwhile. In this way, any problems with the app's migration can be identified and addressed before another block of migration effort is begun.

As it happens, configuring EF 6 in the eShop sample migration doesn't require any special work, since this work was performed in the Autofac ApplicationModule. The only problem is that currently the CatalogDBContext class tries to read its connection string from *web.config*. To address this, the connection details need to be added to *appsettings.json*. Then the connection string must be passed into CatalogDBContext when it's created.

Update the *appsettings.json* to include the connection string. The full file is listed here:

```
{
  "ConnectionStrings": {
    "DefaultConnection":
"Server=(localdb)\\mssqllocaldb;Database=eShopPorted;Trusted_Connection=True;MultipleActive
ResultSets=true"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*",
  "UseMockData": "false",
  "UseCustomizationData": "true"
}
```

The connection string must be passed into the constructor when the DbContext is created. Since the instances are created by Autofac, the change needs to be made in ApplicationModule. Modify the module to take in a connectionString in its constructor and assign it to a field. Then modify the registration for CatalogDBContext to add connection string as a parameter:

```
builder.RegisterType<CatalogDBContext>()
  .WithParameter("connectionString", _connectionString)
  .InstancePerLifetimeScope();
```

The parameter must also be added to a new constructor overload in CatalogDBContext itself:

```
public CatalogDBContext(string connectionString) : base(connectionString)
{
}
```

Finally, ConfigureServices must read the connection string from Config and pass it into the ApplicationModule when it instantiates it:

```
bool useMockData = Configuration.GetValue<bool>("UseMockData");
string connectionString = Configuration.GetConnectionString("DefaultConnection");
builder.RegisterModule(new ApplicationModule(useMockData, connectionString));
```

With this code in place, the app runs as it did before, connecting to a SQL Server database when UseMockData is false.

The app can be deployed and run in production at this point, converted to ASP.NET Core but still running on .NET Framework and EF 6. If desired, the app can be migrated to run on .NET Core and Entity Framework Core, which will bring additional advantages described in earlier chapters. Specific to Entity Framework, this documentation compares EF Core and EF 6 and includes a grid showing which library supports each of dozens of individual features.

Migrate to Entity Framework Core

Assuming a decision is made to migrate to EF Core, the steps can be fairly straightforward, especially if the original app used a code-based model approach. When preparing to port from EF 6 to EF Core, review the availability of features in the destination version of EF Core you'll be using. Review the documentation on porting from and EDMX-based model versus porting from a code-based model.

To upgrade to EF Core 2.2, the basic steps involved are to add the appropriate NuGet package(s) and update namespaces. Then adjust how the connection string is passed to the DbContext type and how they're wired up for dependency injection.

EF Core is added as a package reference to the project:

```
<PackageReference Include="Microsoft.EntityFrameworkCore" Version="2.2.6" />
```

The reference to EF 6 is removed:

```
<PackageReference Include="EntityFramework" Version="6.2.0" />
```

The compiler will report errors in CatalogDBContext and CatalogDBInitializer. CatalogDbContext needs to have the old namespaces removed and replaced with Microsoft.EntityFrameworkCore. Its constructors can be removed. DbModelBuilder should be replaced with ModelBuilder. The helper methods for configuring types are moved to separate classes implementing IEntityTypeConfiguration<T>. Then the CatalogDBContext class's OnModelCreating method simply becomes:

```
protected override void OnModelCreating(ModelBuilder builder)
{
    builder.ApplyConfigurationsFromAssembly(Assembly.GetExecutingAssembly());

    base.OnModelCreating(builder);
}
```

Other changes involved include:

- HasDatabaseGeneratedOption(DatabaseGeneratedOption.None) replaced with ValueGeneratedNever()

- HasRequired<T> replaced with HasOne<T>
- Installed Microsoft.EntityFrameworkCore.Relational package
- Add a constructor to CatalogDBContext taking DbContextOptions and passing it to the base constructor

An example configuration class for CatalogType is shown here:

```csharp
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace eShopPorted.Models.Config
{
    public class CatalogTypeConfig : IEntityTypeConfiguration<CatalogType>
    {
        public void Configure(EntityTypeBuilder<CatalogType> builder)
        {
            builder.ToTable(nameof(CatalogType));

            builder.HasKey(ci => ci.Id);

            builder.Property(ci => ci.Id)
                .IsRequired();

            builder.Property(cb => cb.Type)
                .IsRequired()
                .HasMaxLength(100);
        }
    }
}
```

The CatalogDBInitializer and its base class, CreateDatabaseIfNotExists<T>, are incompatible with EF Core. The purpose of this class is to create and seed the database. Using EF Core will create and drop the associated database for a DbContext using these methods:

```
dbContext.Database.EnsureDeleted();
dbContext.Database.EnsureCreated();
```

Seeding data in EF Core can be done with manual scripts, or as part of the type configuration. Along with other entity properties, seed data can be configured in IEntityTypeConfiguration classes by using builder.HasData(). The original app loaded seed data from CSV files in the *Setup* directory. Given that there are only a handful of items, these data records can instead be added as part of the entity configuration. This approach works well for lookup data in tables that change infrequently. Adding the following to CatalogTypeConfig's Configure method ensures the associated rows are present when the database is created:

```csharp
builder.HasData(
    new CatalogType { Id = 1, Type = "Mug" },
    new CatalogType { Id = 2, Type = "T-Shirt" },
    new CatalogType { Id = 3, Type = "Sheet" },
    new CatalogType { Id = 4, Type = "USB Memory Stick" }
);
```

The initial app includes a PreconfiguredData class, which includes data for CatalogBrand and CatalogType, so using this method the HasData call reduces to:

```
builder.HasData(
    PreconfiguredData.GetPreconfiguredCatalogBrands()
);
```

The CatalogItem data can also be pulled from PreconfiguredData, and assuming the associated images are kept in source control, that is the last table needed for the app to function. The CatalogDBInitializer class can be removed, along with any references to it. The CatalogItemHiLoGenerator class and the SQL files in the Infrastructure directory are also removed, along with any references to them (in CatalogService, ApplicationModule).

With the elimination of the special key generator classes for CatalogItem, this code now is removed from CatalogItemConfig:

```
builder.Property(ci => ci.Id)
    .ValueGeneratedNever()
    .IsRequired();
```

With these modifications, the ASP.NET Core app builds, but it doesn't yet work with EF Core, which must still be configured for dependency injection. With EF Core, the simplest way to configure it is in ConfigureServices:

```
public IServiceProvider ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    bool useMockData = Configuration.GetValue<bool>("UseMockData");
    if (!useMockData)
    {
        string connectionString = Configuration.GetConnectionString("DefaultConnection");

        services.AddDbContext<CatalogDBContext>(options =>
            options.UseSqlServer(connectionString)
        );
    }

    // Create Autofac container builder
    var builder = new ContainerBuilder();
    builder.Populate(services);
    builder.RegisterModule(new ApplicationModule(useMockData));

    ILifetimeScope container = builder.Build();

    return new AutofacServiceProvider(container);
}
```

The final version of Autofac's ApplicationModule only configures one type, depending on whether the app is configured to use mock data:

```
public class ApplicationModule : Module
{
    private bool _useMockData;

    public ApplicationModule(bool useMockData)
    {
        _useMockData = useMockData;
    }

    protected override void Load(ContainerBuilder builder)
```

```
    {
        if (_useMockData)
        {
            builder.RegisterType<CatalogServiceMock>()
                .As<ICatalogService>()
                .SingleInstance();
        }
        else
        {
            builder.RegisterType<CatalogService>()
                .As<ICatalogService>()
                .InstancePerLifetimeScope();
        }
    }
}
```

The ported app runs, but doesn't display any data if configured to use non-mock data. The seed data added through HasData is only inserted when migrations are applied. The source app didn't use migrations, and if it had, they wouldn't migrate as-is. The best approach is to start with a new migration script. To do this, add a package reference for Microsoft.EntityFrameworkCore.Design and open a terminal window in the project root. Then run:

```
dotnet ef migrations add Initial
```

Drop the existing *eShopPorted* database if it exists, then run:

```
dotnet ef database update
```

This creates and seeds the database. It's now ready to run, with a few small updates left to address.

Fix all TODO tasks

Running the ported app at this point reveals that no pictures are shown on the page. This is because the PictureUri property of CatalogItem is never set. Looking at the list of TODO items we created using Visual Studio's **Task List**, the only one that remains is in CatalogController, with a note to "Reference pic from wwwroot." The code in question is:

```
private void AddUriPlaceHolder(CatalogItem item)
{
    //TODO: Reference pic from wwwroot
    //item.PictureUri = this.Url.RouteUrl(PicController.GetPicRouteName, new {
catalogItemId = item.Id }, this.Request.Url.Scheme);
}
```

The simplest fix is to reference the public image files in the site's public *wwwroot/Pics* directory. This task can be accomplished by replacing the method with the following code:

```
private void AddUriPlaceHolder(CatalogItem item)
{
    item.PictureUri = $"/Pics/{item.Id}.png";
}
```

With this change, running the app reveals the images work as before.

Additional MVC customizations

The *eShopLegacyMVC* app is fairly simple, so there isn't much to configure in terms of default MVC behavior. However, if you do need to configure additional MVC components, such as CORS, filters, and route constraints, you generally provide this information in Startup.ConfigureServices, where UseMvc is called. For example, the following code listing configures CORS and sets up a global action filter:

```csharp
public void ConfigureServices(IServiceCollection services)
{
    services.AddCors(options =>
    {
        options.AddPolicy(MyAllowSpecificOrigins,
            builder =>
                builder.WithOrigins("http://example.com", "http://www.contoso.com")
                    .AllowAnyHeader()
                    .AllowAnyMethod());
    });

    services.AddMvc(options =>
    {
      options.Filters.Add(new SampleGlobalActionFilter());
    }).SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

> **Note**
>
> To finish configuring CORS, you must also call app.UseCors() in Configure.

Other advanced scenarios, like adding custom model binders, formatters, and more are covered in the detailed ASP.NET Core docs. Generally these can be applied on an individual controller or action basis, or globally using the same options approach shown in the previous code listing.

Other dependencies

Dependencies that use .NET Framework features that had a dependency on the legacy configuration model, such as the WCF client type and tracing code, must be modified when ported. Rather than having these types pull in their configuration information directly, they should be configured in code. For example, a connection to a WCF service that was configured in an ASP.NET app's *web.config* to use basicHttpBinding could instead be configured programmatically with the following code:

```csharp
var binding = new BasicHttpBinding();
binding.MaxReceivedMessageSize = 2_000_000;

var endpointAddress = new EndpointAddress("http://localhost:9200/ExampleService");

var myClient = new MyServiceClient(binding, endpointAddress);
```

Rather than relying on config files for its settings, WCF clients and other .NET Framework types should have their settings specified in code. Configured in this manner, these types can continue to work in ASP.NET Core 2.2 apps.

References

- [eShopModernizing GitHub repository](#)
- [Your API and ViewModels Should Not Reference Domain Models](#)
- [Developer Exception Page Middleware](#)
- [Deep Dive into EF Core HasData](#)

Deployment scenarios when migrating to ASP.NET Core

Existing ASP.NET MVC and Web API apps run on IIS and Windows. Large apps may require a phased or side-by-side approach when porting to ASP.NET Core. In previous chapters, you learned a number of strategies for migrating large .NET Framework apps to ASP.NET Core in phases. In this chapter, you will see how different deployment scenarios can be achieved when there is a need to maintain the original app in production while migrating portions of it.

Split a large web app

Consider the common scenario of a large web app that currently is hosted on IIS in a single web site. Within the large app, functionality is segmented into different routes and/or directories. The app is a mix of MVC views and API endpoints. The MVC routes include many different paths based on functionality and all start from the root of the app using the standard /{controller}/{action}/{id?} route template. The API endpoints follow a similar pattern, but are all under an /api root.

Assuming the task of porting the app is split such that either the MVC functionality or the API functionality is migrated to ASP.NET Core first, how would the original site continue to function seamlessly with the new ASP.NET Core app running somewhere else? Users of the system should continue to see the same URLs they did prior to the migration, unless it's absolutely necessary to change them.

Fortunately, IIS is a very feature-rich web server, and two features it has had for some time are its URL Rewrite module and Application Request Routing. Using these features, IIS can act as a reverse proxy, routing client requests to the appropriate back-end web app. To configure IIS as a reverse proxy, check the **Enable proxy** checkbox in the Application Request Routing feature, then add a URL Rewrite rule like this one:

```
<rule name="NetCoreProxy">
  <match url="(.*)> />
  <action type="Rewrite" url="http://servername/{R:1}" />
</rule>
```

As a reverse proxy, IIS can route traffic matching certain patterns to entirely separate apps, potentially on different servers.

Using just the URL Rewrite module (perhaps combined with host headers), IIS can easily support multiple web sites, each potentially running different versions of .NET. A large web app might be deployed as a collection of individual sites, each responding to different IP addresses and/or host headers, or as a single web site with one or more sub-applications in it responding to certain URL paths (which doesn't even require URL Rewrite).

> **Important**
>
> Subdomains typically refer to the portion of a domain preceding the top two levels. For example, in the domain api.contoso.com, api is a subdomain of the contoso.com domain (which itself is composed of the contoso domain name and the .com top-level domain or TLD). URL paths refer to portion of the URL that follows the domain name, starting with a /. The URL https://contoso.com/api has a path of /api.

There are pros and cons to using the same or different subdomains (and domains) to host a single app. Features like cookies and intra-app communication using mechanisms like CORS may require more configuration to work properly in distributed apps. However, apps that use different subdomains can more easily use DNS to route requests to entirely different network destinations, and so can more easily be deployed to many different servers (virtual or otherwise) without the need for IIS to act as a reverse proxy.

In the example described above, assume the API endpoints are designated as the first part of the app to be ported to ASP.NET Core. In this case, a new ASP.NET Core app is created and hosted in IIS as a separate web *application* within the existing ASP.NET MVC web *site*. Since it will be added as a child of the original web site and will be named *api*, its default route should no longer begin with api/. Keeping this would result in it matching URLs of the form /api/api/endpoint.

Figure 5-1 shows how the ASP.NET Core 2.1 *api* app appears in IIS Manager as a part of the existing *DotNetMvcApp* site.
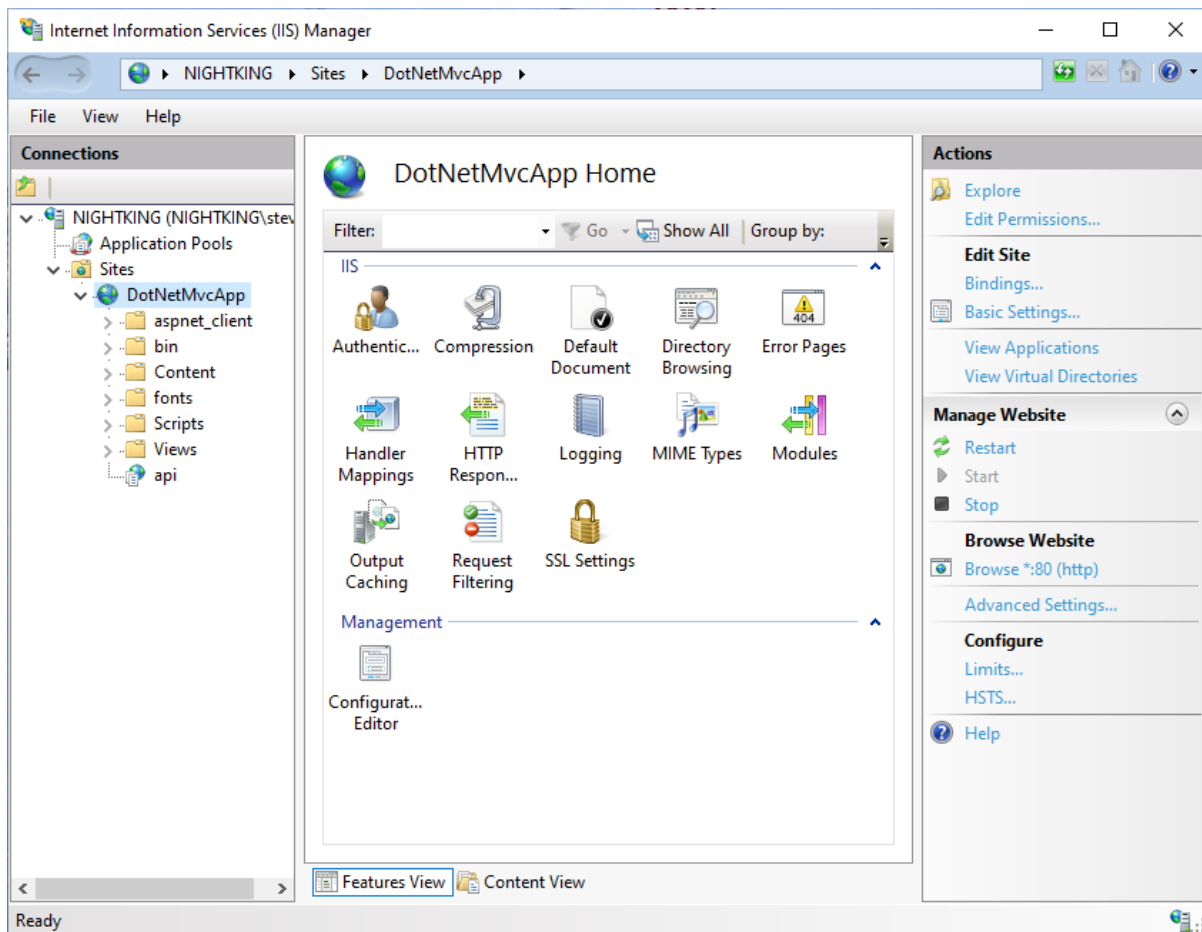
Figure 5-1. .NET Framework Site with .NET Core app in IIS.

The *DotNetMvcApp* site is hosted as an MVC 5 app running on .NET Framework 4.7.2. It has its own IIS app pool configured in integrated mode and running .NET CLR version 4.0.30319. The *api* app is an ASP.NET Core app running on .NET Framework 4.6.1 (net461). It was added to the *DotNetMvcApp* as a new IIS app and configured to use its own Application Pool. Its Application Pool is also running in integrated mode but is configured with a .NET CLR version of **No Managed Code** since it will be executed using the ASP.NET Core Module. The version of the ASP.NET Core app is just an example. It could also be configured to run on .NET Core 3.1 or .NET 5.0. Though at that point, it would no longer be able to target .NET Framework libraries (see Choose the Right .NET Core Version)

Configured in this manner, the only change that must be made in order for the ASP.NET Core app's APIs to be routed properly is to change its default route template from [Route("[api/controller]")] to [Route("[controller]")].

Alternately the ASP.NET Core app can be another top-level web site in IIS. In this case, you can configure the original site to use a rewrite rule (with URL Rewrite) that will redirect to the other app if the path starts with /api. The ASP.NET Core app can use a different host header for its route so that it doesn't conflict with the main app but can still respond to requests using root-based routes.

As an example, the same ASP.NET Core app used in Figure 5-1 can be deployed to another folder configured as an IIS web site. The site should use an app pool configured just as before, with **No**

Managed Code. Configure its bindings to respond to a unique host name on the server, such as api.contoso.com. To configure URL Rewrite to rewrite requests matching /api just add a new inbound rule at the IIS server (or individual site) level. Match the pattern ^/api(.*) and specify an Action type of Rewrite and a Rewrite URL of api.contoso.com/{R:1}. The combination of using (.*) in the matching pattern and {R:1} in the rewrite URL will ensure the rest of the path gets used with the new URL. With this in place, separate sites on the same IIS server can coexist running separate versions of .NET, but they can be made to appear to the Internet as one web app. Figure 5-2 shows the rewrite rule as configured in IIS with the separate web site.
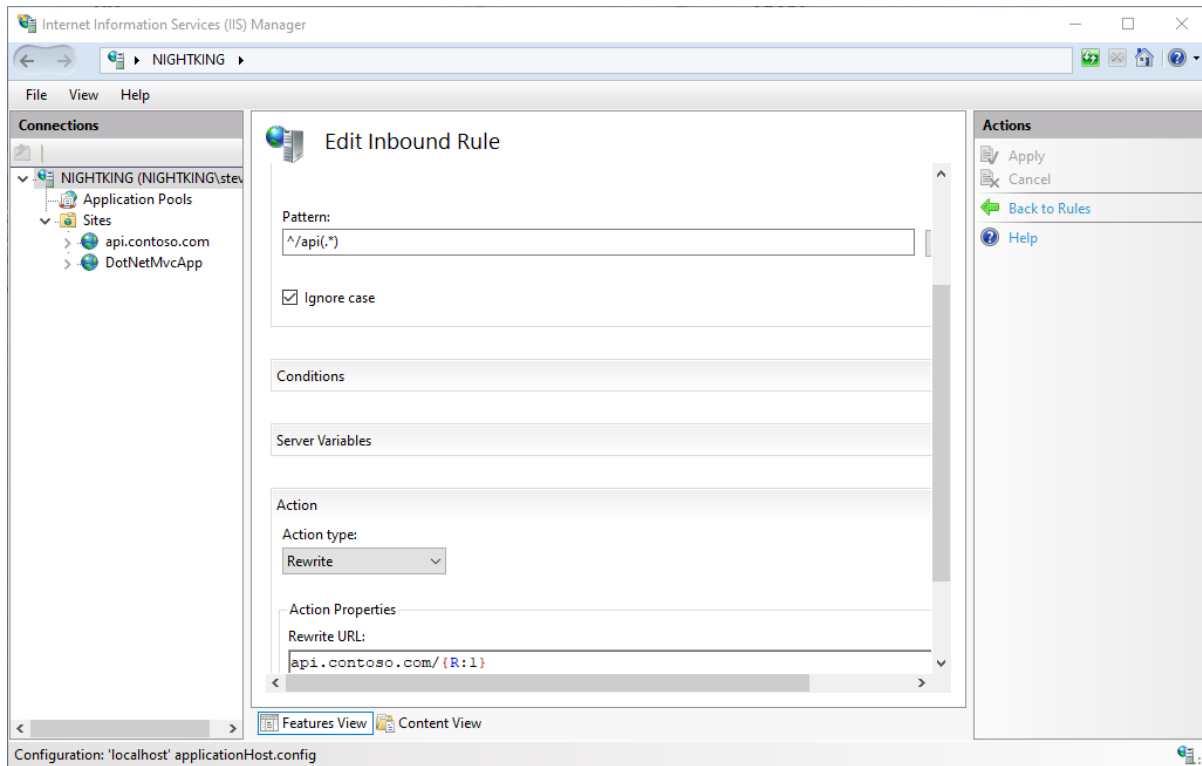

Figure 5-2. Rewrite rule to rewrite subfolder requests to another web site.

If your app requires single sign-on between different sites or apps within IIS, refer to the documentation on how to share authentication cookies among ASP.NET apps for detailed instructions on supporting this scenario.

Summary

A common approach to porting large apps from .NET Framework to ASP.NET Core is to choose individual portions of the app to migrate one by one. As each piece of the app is ported, the entire app remains running and usable, with some parts of it running in its original configuration and other parts running on some version of .NET Core. By following this approach, a large app migration can be performed incrementally. This approach results in limiting risk by providing more rapid feedback and reducing total surface area involved in testing. It also allows for more rapid realization of benefits of .NET Core, such as performance increases. Although ASP.NET Core apps are no longer required to be hosted on IIS, IIS remains a very flexible and powerful web server that can be configured to support a

variety of hosting scenarios involving both .NET Framework and ASP.NET Core apps on the same IIS instance or even hosted on different servers.

References

- [Host ASP.NET Core on Windows with IIS](#)
- [URL Rewrite module and Application Request Routing](#)
- [URL Rewrite](#)
- [ASP.NET Core Module](#)
- [Share authentication cookies among ASP.NET apps](#)

Summary

In this book, you've been given the resources needed to decide whether it makes sense to port your organization's existing ASP.NET apps running on .NET Framework to ASP.NET Core. You've learned about important considerations for choosing when it makes sense to migrate to .NET Core, and when it may be appropriate to keep (parts of) your app on .NET Framework. There are differences between .NET Core versions and their capabilities and compatibilities with .NET Framework, and you learned how to choose the right version of .NET Core for your app.

Porting a large app often entails a fair amount of risk and effort. You learned how to mitigate this risk by employing one or more incremental migration strategies along with several deployment strategies for keeping partially migrated apps running in production.

There are many architectural differences between ASP.NET and ASP.NET Core. In chapter 2, you learned about many of these differences and how they relate to your app's migration. This chapter covered everything from app startup and low-level middleware to high-level controller and Web API differences and new features enabling much better testing scenarios.

Large apps are often comprised of many projects and packages, and dependencies can play a major role in determining how easy or difficult migration may be. Chapter 3 helped you identify the sequence in which to migrate projects and how to understand and update your app's dependencies. It also detailed additional strategies for migrating apps while keeping them running in production.

In chapter 4, you saw how a real ASP.NET MVC reference app was migrated to ASP.NET Core. This chapter included a detailed breakdown of all the changes that were needed to take the existing app and port it over to run on ASP.NET Core. Refer back to it if you have specific questions about the porting process and some of its more specific details.

Finally, chapter 5 detailed specific deployment scenarios focused on IIS. You saw how you can use your existing IIS web server to host parts of your app that have been ported to ASP.NET Core while keeping the app's public URLs consistent. IIS includes great support for URL rewriting and request routing that enables it to host multiple versions of your site side by side or even on different servers, with no change to the public-facing URLs the app exposes.