.NET 7 AND C# 11

SUCCINCTLY®

BY DIRK STRAUSS

SUCCINCTLY® EBOOK SERIES

Syncfusion®

.NET 7 and C# Succinctly®

Dirk Strauss

Foreword by Daniel Jebaraj

ISBN: 978-1-64200-240-9

Technical Reviewer: James McCaffrey

Copy Editor: Courtney Wright

Acquisitions Coordinator: Tres Watkins, VP of content, Syncfusion, Inc.

Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

Table of Contents

The *Succinctly*® Series of Books

Daniel Jebaraj
CEO of Syncfusion®, Inc.

When we published our first *Succinctly*® series book in 2012, *jQuery Succinctly*®, our goal was to produce a series of concise technical books targeted at software developers working primarily on the Microsoft platform. We firmly believed then, as we do now, that most topics of interest can be translated into books that are about 100 pages in length.

We have since published over 200 books that have been downloaded millions of times. Reaching more than 1.7 million readers around the world, we have more than 70 authors who now cover a wider range of topics, such as Blazor, machine learning, and big data.

Each author is carefully chosen from a pool of talented experts who share our vision. The book before you and the others in this series are the result of our authors' tireless work. Within these pages, you will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

We are absolutely thrilled with the enthusiastic reception of our books. We believe the *Succinctly* series is the largest library of free technical books being actively published today. Truly exciting!

Our goal is to keep the information free and easily available so that anyone with a computing device and internet access can obtain concise information and benefit from it. The books will always be free. Any updates we publish will also be free.

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctlyseries@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on social media and help us spread the word about the *Succinctly*® series!

About the Author

As a seasoned software developer with over 17 years of experience utilizing C# and Visual Studio, I have had the privilege of working with a number of companies and learning from some of the most talented individuals in the industry. In addition to my professional experience, I have published multiple books on topics such as C#, Visual Studio, and ASP.NET Core. My passion for programming is unwavering, and I am dedicated to staying current with the latest technology and sharing my expertise with others.

Chapter 1 Introducing .NET 7

To understand how we arrived at .NET 7, we should first briefly examine the history of .NET in general.

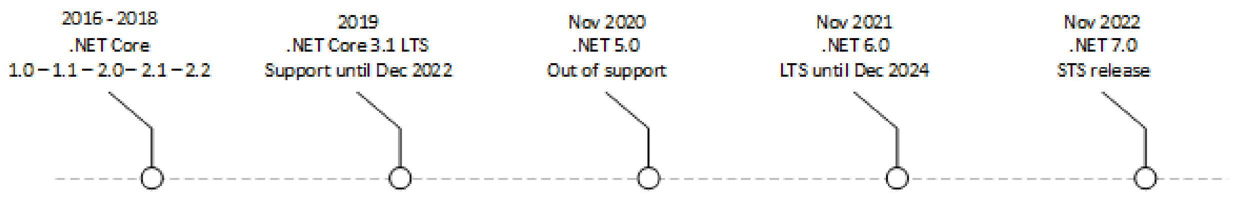


Figure 1: The History of .NET

.NET Core 1.0 was released in 2016 and heralded the start of what we know today as .NET 7. Before this release, developers were accustomed to the .NET Framework first released in February 2002. It was closed-source and only ran on Windows. .NET Core, however, was the first .NET version to run across platforms and is an open-source project.

.NET Core saw many releases between 2016 and 2018, culminating in the release of .NET Core 3.1 in 2019. .NET Core 3.1 was a long-term support version, with support for it ending in December 2022.

Many companies developed software on .NET Core 3.1 due to its long-term support status. The other day, I ran into a project developed on .NET Core 3.1 that I had to upgrade to .NET 7. It was, therefore, a very popular version of .NET Core. Then, in 2020, Microsoft surprised us all with the release of .NET 5.0. (version 4 was skipped to avoid confusion with the .NET Framework versions.) Microsoft touted .NET 5.0 as a single, unified platform that would replace .NET Core, Xamarin, and the .NET Framework. Support for .NET 5.0 ended in May 2022.

In November 2021, Microsoft released .NET 6.0, with long-term support until December 2024. This introduced a host of new features and unified the various .NET platforms into a single .NET version.

The standard-term support release of .NET 7.0 in November 2022 brings us to where we are today: a unified .NET that supports cross-platform application development, allowing developers to create a host of different applications—from desktop apps to cloud apps, web applications, Unity games, mobile applications, and AI apps.

The unification of .NET means that we now have a single BCL (or base class library) and SDK. With the release of MAUI, we can create cross-platform native UI applications.

About .NET 7

Compared to .NET 6.0, there were fewer changes with the release of .NET 7. It did, however, bring with it performance increases. A thousand performance-impacting pull requests went into the runtime and core libraries, resulting in developers often gaining a significant performance boost in their applications by simply upgrading to .NET 7.0. These performance improvements make .NET 7.0 the fastest version of .NET thus far.

This, alone, is reason enough to upgrade to .NET 7.0. New APIs have also been introduced in .NET 7.0. Changes include:

- Microseconds and nanoseconds were added to DateTime.
- Changes have been made to the Microsoft.Extensions.Caching namespace.
- Support for working with Tar (tape archive) files has been included with System.Formats.Tar.
- Various options were also added to System.Text.Json.

ASP.NET Core also had notable changes. These include additions such as rate limiting, allowing ASP.NET Core applications to limit the requests they handle in a specific amount of time. Output caching has also been added and configures how responses from the ASP.NET Core application are cached.

Minimal APIs, introduced in .NET 6.0, also received an update, including endpoint filters and route groups.

Introduced in early 2022, MAUI has also received some love. MAUI allows developers to build mobile applications targeting iOS and Android, Windows desktop, and macOS. With .NET 7.0, one of the key improvement areas to MAUI has been performance. MAUI also received some new controls, such as the map control. Desktop targeting has also been improved.

Some of the tooling improvements include Azure support, container support, hot reload, and CLI improvements. Microsoft has also made improvements to the upgrade assistant, allowing developers to upgrade their applications to the latest version of .NET. While it is by no means perfect, it significantly assists developers with going through the pain of upgrading older .NET applications.

.NET 7.0 is a standard-term support release, and it is supported for six months after the release of .NET 8.0. The result is that any production application running .NET 7.0 must be upgraded when .NET 8.0 is released in November 2023.

Upgrading existing applications to .NET 7

By now, it should be clear that moving to .NET 7.0 is wise if you want to use the latest features, let alone utilize the speed enhancements that .NET 7.0 brings.

Luckily for us, upgrading an existing .NET 6.0 project to .NET 7.0 is quite simple, as illustrated in the following code listings.

If you view the project file of an application built on .NET 6.0, you see that the **`TargetFramework`** element is set to the value **`net6.0`**.

Code Listing 1: A .NET 6.0 Project File

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

Modify the project file to reference **`net7.0`** to make use of .NET 7.0.

Code Listing 2: A .NET 7.0 Project File

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net7.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

Remember to upgrade any other NuGet packages to their .NET 7.0 counterparts, also.

It is worth noting that since .NET 6.0, the **`Startup`** class has been removed, and the Program.cs file also now uses top-level statements. These changes are optional, however. (I guess I'm an old man, because I am not too fond of these changes and still favor having a **`Startup`** class.)

It's also rather apparent that the older the project that you are upgrading, the more work you have to do to upgrade it. Upgrading a project developed with .NET Core 1.x, for example, requires much more effort, but it is certainly doable.

Tip: It is worth noting that using the .NET Upgrade Assistant is an excellent choice for larger projects. You can download the .NET Upgrade Assistant from this URL.

Unfortunately, if your project is built on the .NET Framework, you have much more work to do. There is no direct upgrade path, meaning you must create a new project and move over the files manually.

Let's have a closer look at the .NET Upgrade Assistant.

The .NET Upgrade Assistant

The .NET Upgrade Assistant is an upgrade tool that assists developers in upgrading their applications to the latest version of .NET. It can also migrate your projects from older platforms, such as Xamarin Forms and UWP, to newer offerings. To get started with the .NET Upgrade Assistant, you can install the Visual Studio Extension or install it from the .NET global tool. Let's look at the Visual Studio Extension first.

Installing the Visual Studio extension

Installing the Visual Studio Extension is done from within Visual Studio. From the **Extensions** menu, click **Manage Extensions**.

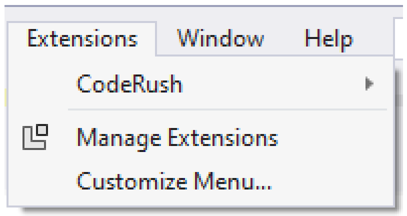
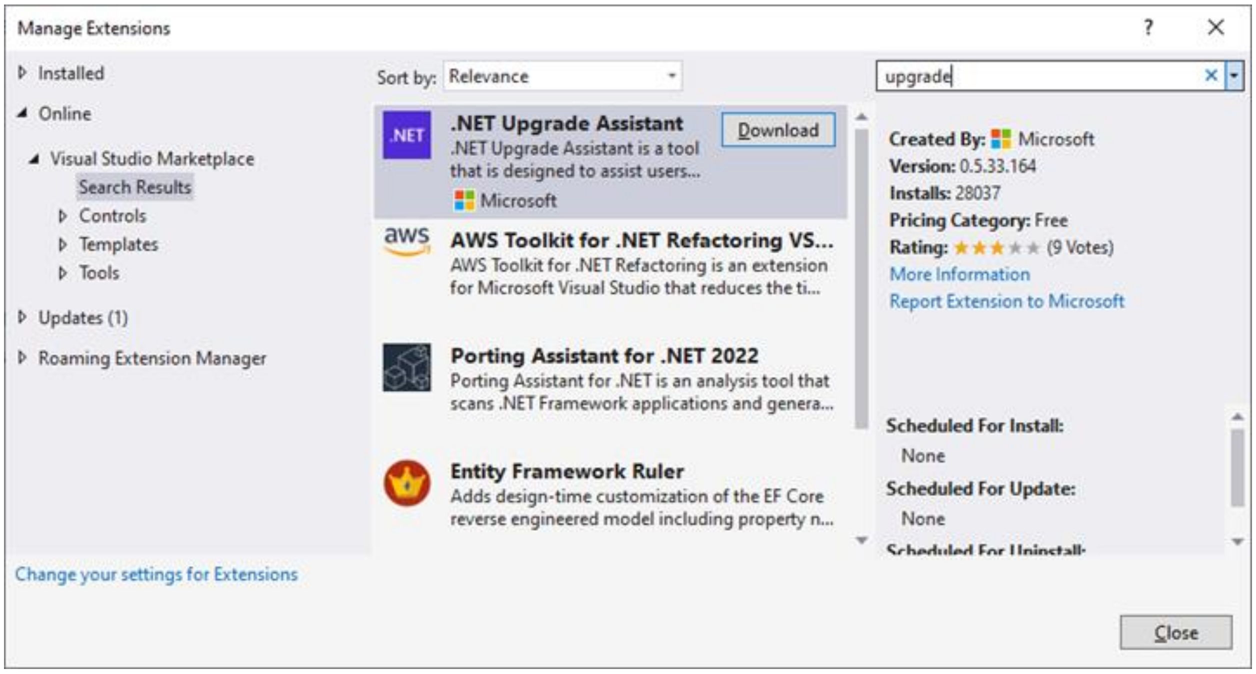


Figure 2: Manage Extensions

In the **Manage Extensions** window that is displayed, search for the **.NET Upgrade Assistant**.



Manage Extensions

- Installed
- Online
  - Visual Studio Marketplace
    - Search Results
    - Controls
    - Templates
    - Tools
- Updates (1)
- Roaming Extension Manager

Change your settings for Extensions

Sort by: Relevance

upgrade

.NET Upgrade Assistant — Download
.NET Upgrade Assistant is a tool that is designed to assist users...
Microsoft

AWS Toolkit for .NET Refactoring VS...
AWS Toolkit for .NET Refactoring is an extension for Microsoft Visual Studio that reduces the ti...

Porting Assistant for .NET 2022
Porting Assistant for .NET is an analysis tool that scans .NET Framework applications and genera...

Entity Framework Ruler
Adds design-time customization of the EF Core reverse engineered model including property n...

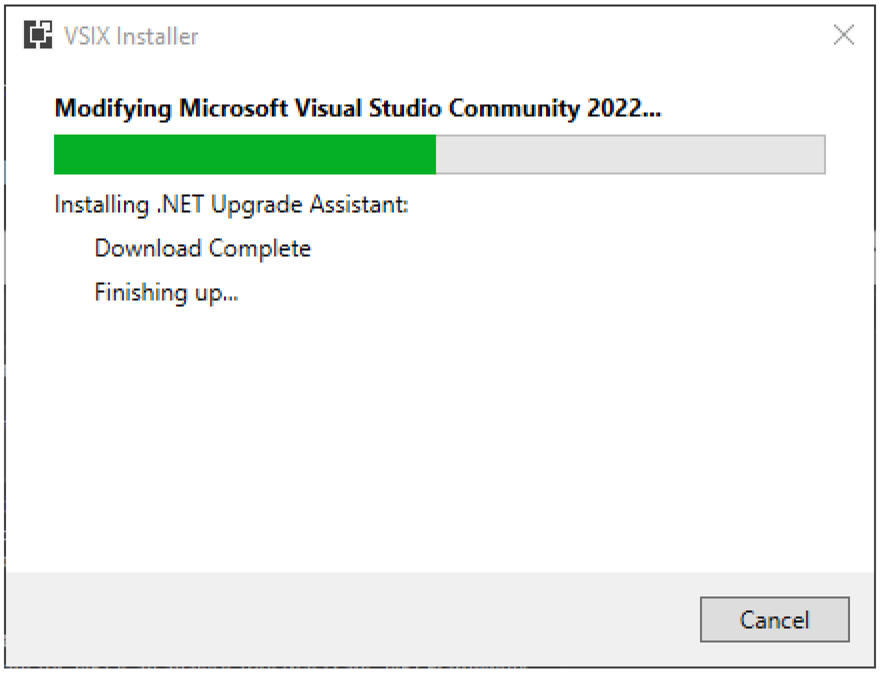
Created By: Microsoft
Version: 0.5.33.164
Installs: 28037
Pricing Category: Free
Rating: ★★★★★ (9 Votes)
More Information
Report Extension to Microsoft

Scheduled For Install:
None
Scheduled For Update:
None
Scheduled For Uninstall:

Close

Figure 3: The Upgrade Assistant Extension

Download the .NET Upgrade Assistant and restart Visual Studio. The process of installing the tool begins, displayed in the VSIX Installer, as seen in Figure 4.


VSIX Installer

Modifying Microsoft Visual Studio Community 2022...

Installing .NET Upgrade Assistant:

Download Complete

Finishing up...

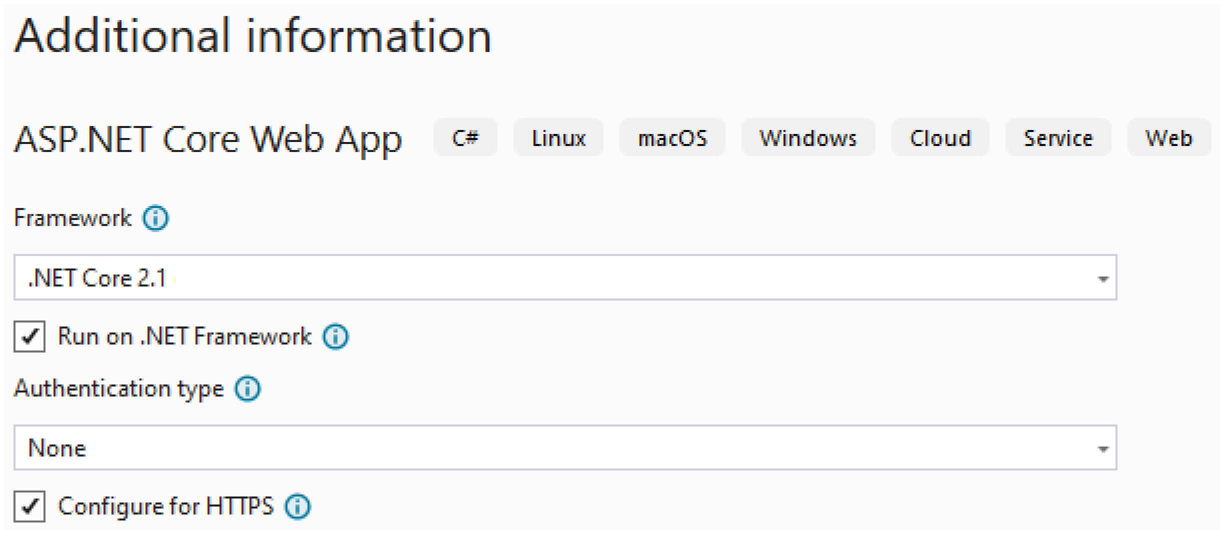
Figure 4: Installing the .NET Upgrade Assistant

Once you install the .NET Upgrade Assistant, you can open your project and start the upgrade process. In my example project, I have a web application for an ice cream parlor's online store that I developed in ASP.NET Core 2.1.

Code Listing 3: My Ice Cream Parlor Project File

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

  <PropertyGroup>
    <TargetFramework>net472</TargetFramework>
    <DebugType>full</DebugType>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore" Version="2.1.7" />
    <PackageReference Include="Microsoft.AspNetCore.CookiePolicy"
Version="2.1.14" />
    <PackageReference Include="Microsoft.AspNetCore.HttpsPolicy"
Version="2.1.1"/>
    <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="2.1.3"
/>
    <PackageReference Include="Microsoft.AspNetCore.StaticFiles"
Version="2.1.1" />
  </ItemGroup>

</Project>
```

You can see that it targets the .NET Framework 4.7.2 because I checked the option to run on .NET Framework when creating the project.


Additional information

ASP.NET Core Web App

Framework: .NET Core 2.1

[✓] Run on .NET Framework

Authentication type: None

[✓] Configure for HTTPS

Figure 5: Additional Information Selected When Creating Project

To start upgrading, I right-click my project in the Solution Explorer and select **Upgrade** from the context menu.

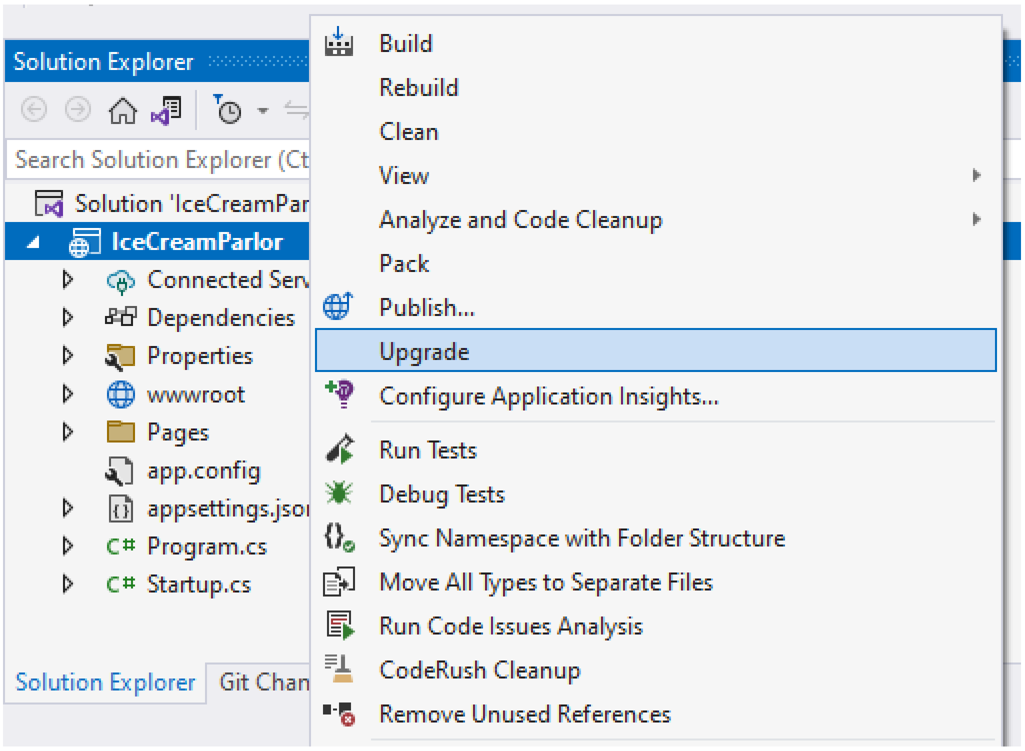


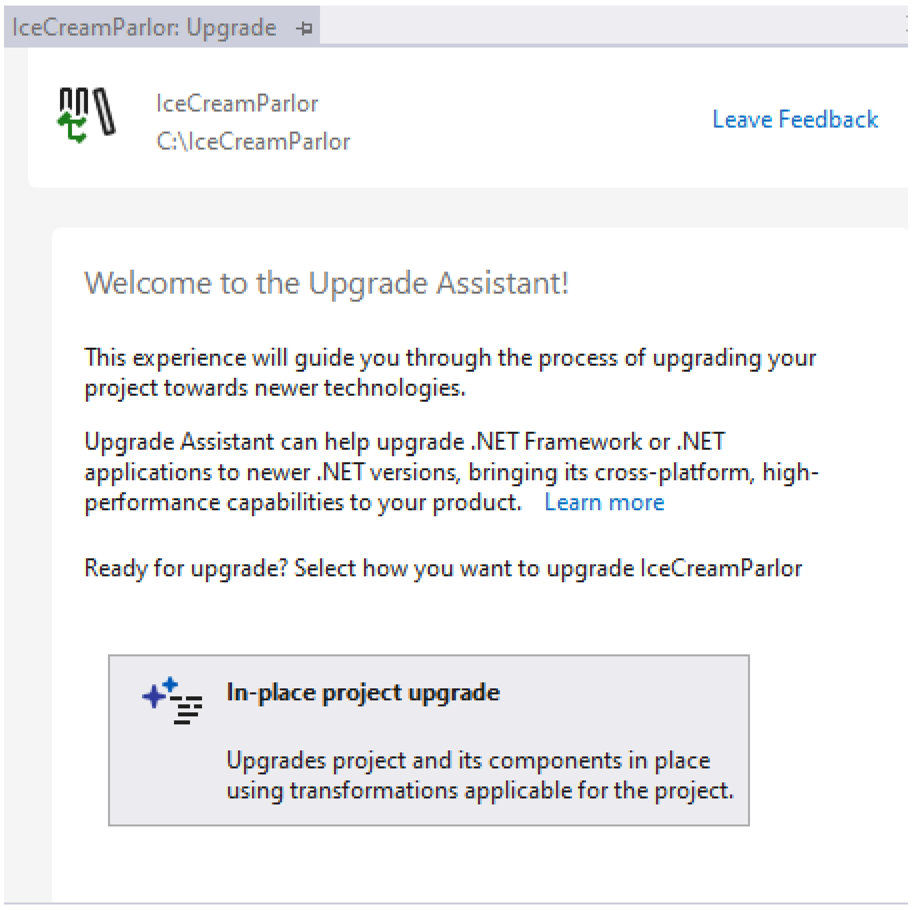
Figure 6: Upgrade a Project

Selecting Upgrade from the context menu displays the Upgrade Assistant and gives me the option of an in-place project upgrade.

The Upgrade Assistant supports three upgrade types. These are:

- In-place
- Side-by-side
- Side-by-side incremental

I will expand a bit more on the three options available later in this chapter. For now, I will click the **In-place project upgrade** option.



IceCreamParlor
C:\IceCreamParlor

Welcome to the Upgrade Assistant!

This experience will guide you through the process of upgrading your project towards newer technologies.

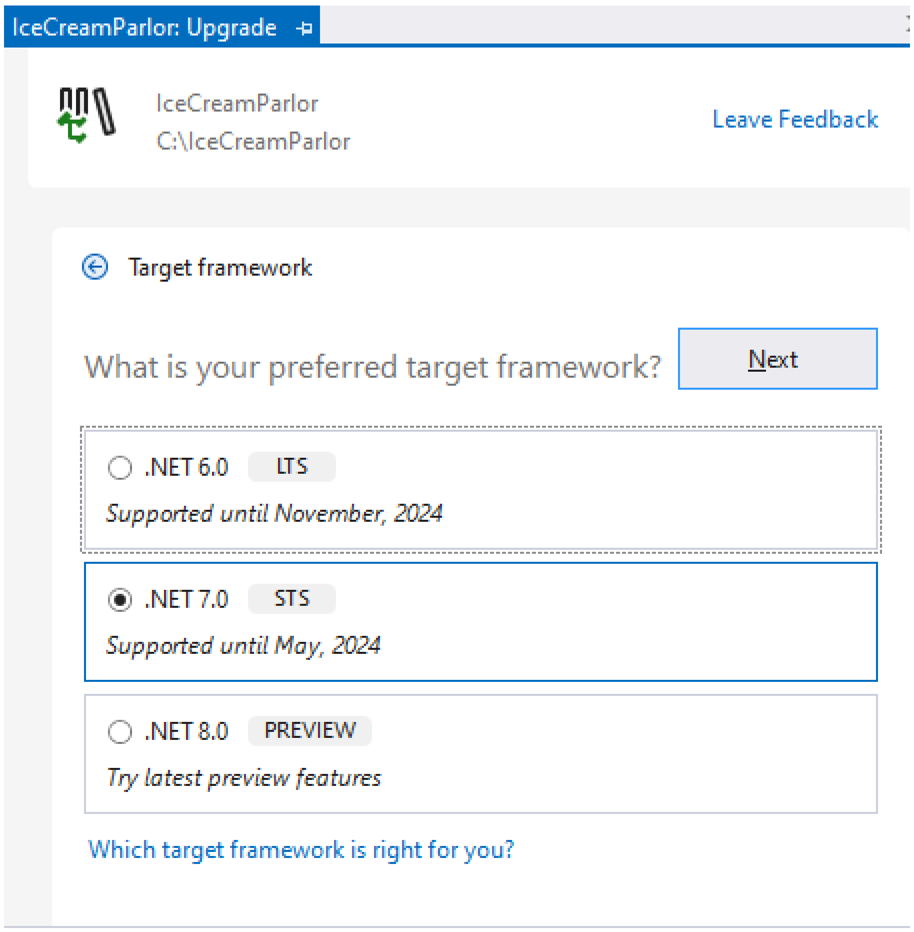
Upgrade Assistant can help upgrade .NET Framework or .NET applications to newer .NET versions, bringing its cross-platform, high-performance capabilities to your product. Learn more

Ready for upgrade? Select how you want to upgrade IceCreamParlor

In-place project upgrade

Upgrades project and its components in place using transformations applicable for the project.

Figure 7: The Upgrade Assistant


IceCreamParlor: Upgrade

IceCreamParlor
C:\IceCreamParlor

Leave Feedback

Target framework

What is your preferred target framework?

Next

( ) .NET 6.0 LTS
Supported until November, 2024

[✓ .NET 7.0 STS]
Supported until May, 2024

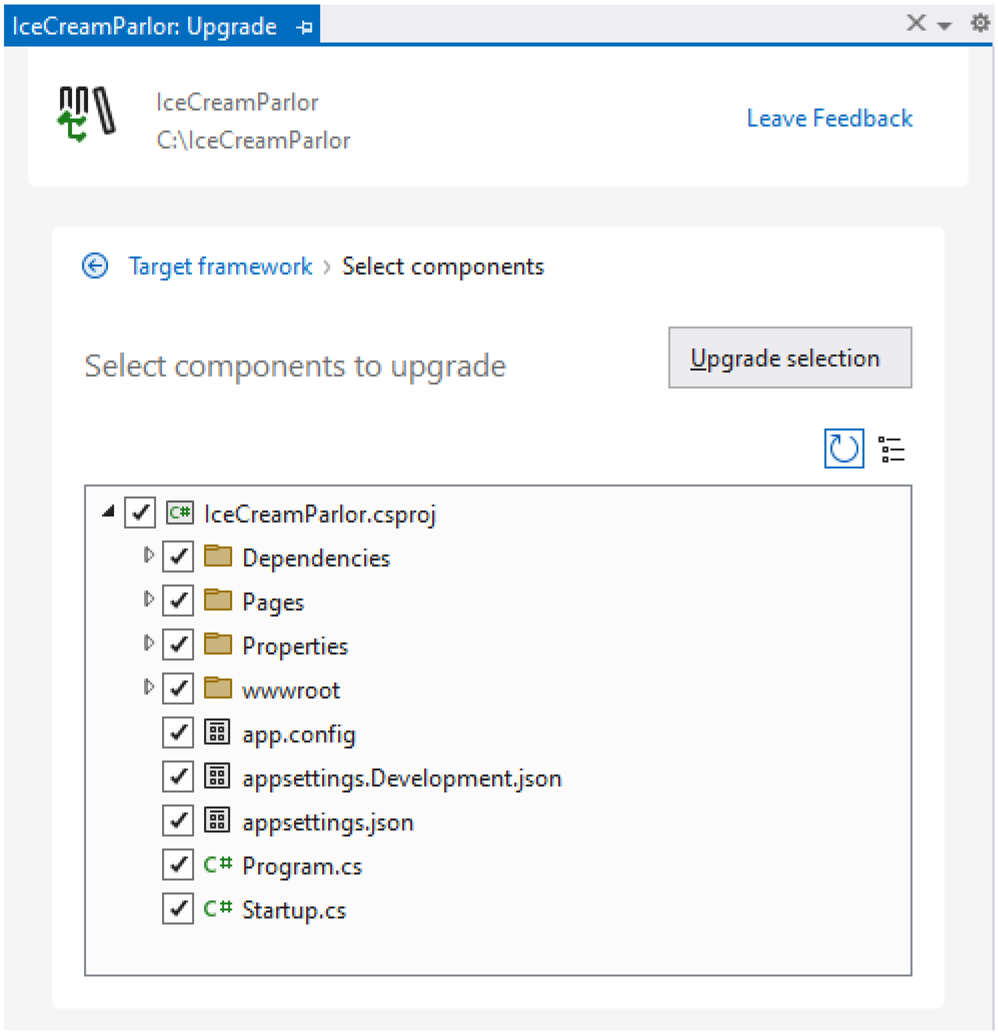
( ) .NET 8.0 PREVIEW
Try latest preview features

Which target framework is right for you?

Figure 8: The Target Framework Selection

The Upgrade Assistant now asks me which target framework I want to upgrade to. As this book is about **.NET 7.0**, I will select it from the available options. In reality, if your project is already in production or due to be released soon, you might want to select .NET 6.0 to take advantage of the long-term support.

Please make no mistake, though: .NET 7.0 is a perfectly reasonable choice and, as mentioned before, brings performance improvements and many other benefits.



- [✓] IceCreamParlor.csproj
  - [✓] Dependencies
  - [✓] Pages
  - [✓] Properties
  - [✓] wwwroot
  - [✓] app.config
  - [✓] appsettings.Development.json
  - [✓] appsettings.json
  - [✓] Program.cs
  - [✓] Startup.cs

Figure 9: The Components to Be Upgraded

The .NET Upgrade Assistant now displays the components it will upgrade. On this screen, you can check and uncheck the components as required. The selections you make here might take some consideration when dealing with large solutions containing many projects.

As my project is small, I keep the default selections and click the **Upgrade selection** button.

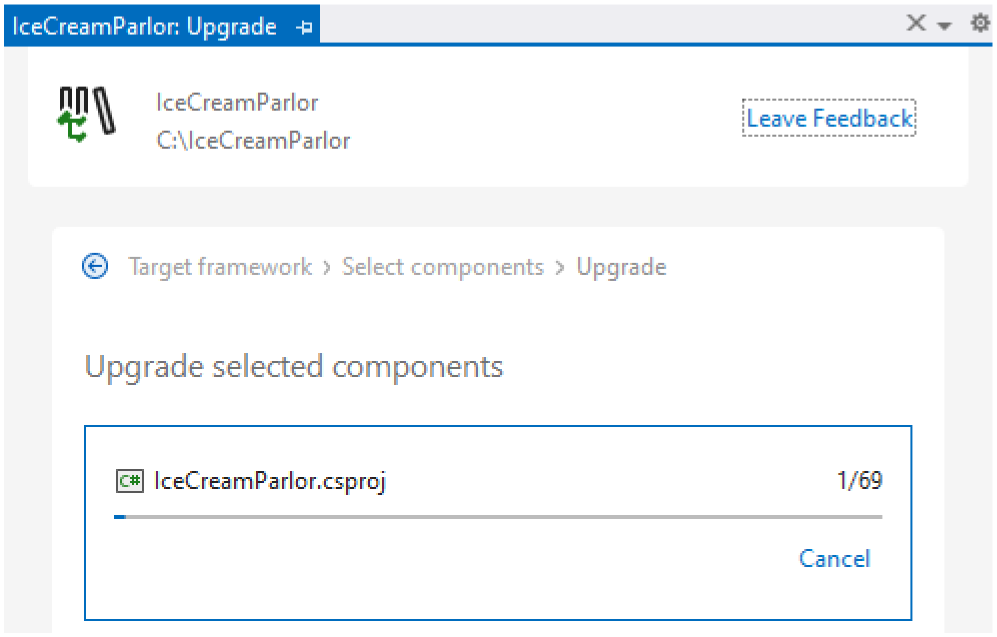
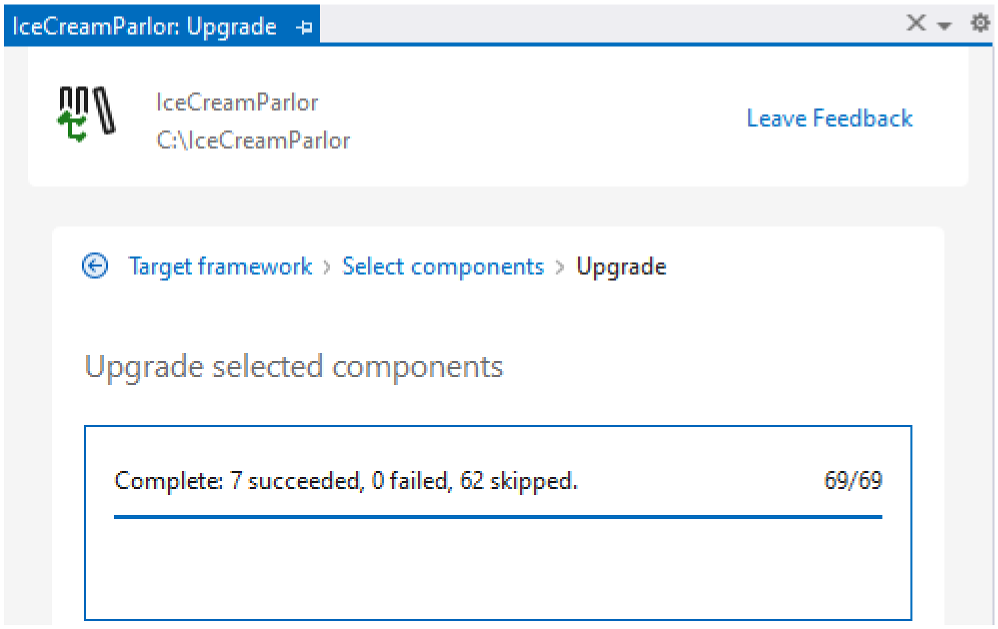


Figure 10: The Upgrade Process

The .NET Upgrade Assistant now starts the upgrade process and displays a progress bar.


IceCreamParlor: Upgrade
IceCreamParlor
C:\IceCreamParlor
Leave Feedback
Target framework › Select components › Upgrade
Upgrade selected components

Complete: 7 succeeded, 0 failed, 62 skipped. 69/69

Figure 11: The Completed Upgrade Process

Once the upgrade process completes, you are shown a summary of the upgrade results with how many components succeeded, failed, or were skipped.

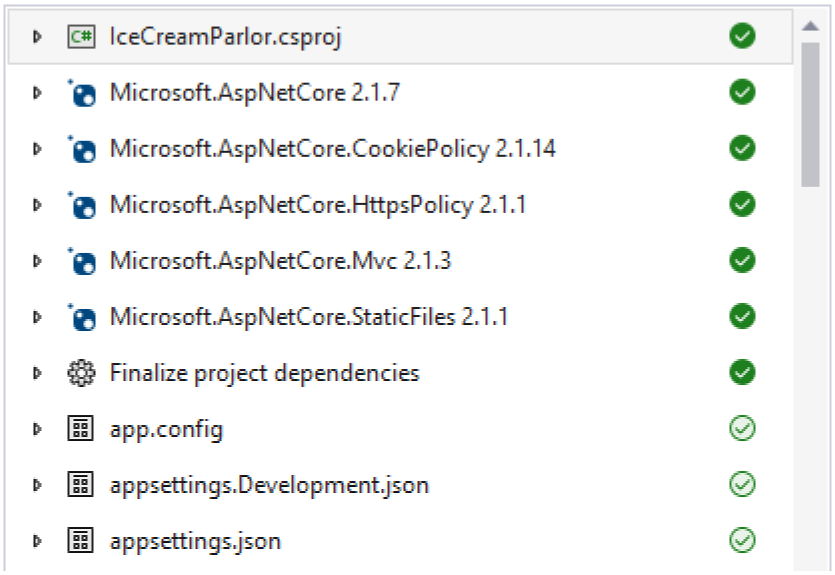

IceCreamParlor.csproj ✓
Microsoft.AspNetCore 2.1.7 ✓
Microsoft.AspNetCore.CookiePolicy 2.1.14 ✓
Microsoft.AspNetCore.HttpsPolicy 2.1.1 ✓
Microsoft.AspNetCore.Mvc 2.1.3 ✓
Microsoft.AspNetCore.StaticFiles 2.1.1 ✓
Finalize project dependencies ✓
app.config ✓
appsettings.Development.json ✓
appsettings.json ✓

Figure 12: The Upgrade Result

You will also see the results listed for each component in the window below the Upgrade Assistant. Hover over each check icon to see a descriptive tooltip.

If you compare the project file from before the upgrade process (Code Listing 3) to the project file after the upgrade process (Code Listing 4), you will notice that a lot has changed.

Code Listing 4: The Upgraded Project File

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net7.0</TargetFramework>
    <DebugType>full</DebugType>
  </PropertyGroup>
</Project>
```

Upgrading your projects using the .NET Upgrade Assistant provides a much quicker experience than manually changing references and package dependencies.

Installing the .NET global tool

Installing the tool is done by running the following command in the .NET CLI.

Code Listing 5: The .NET CLI Command

```
dotnet tool install -g upgrade-assistant
```

Because the .NET Upgrade Assistant is installed as a .NET tool, you can update it easily by running the following command.

Code Listing 6: Updating the .NET Upgrade Assistant

```
dotnet tool update -g upgrade-assistant
```

With the tool installed, navigate to the directory containing your project that needs to be upgraded and call the following command.

Code Listing 7: Upgrading a Visual Studio Project

```
upgrade-assistant upgrade
```

The tool now provides an interactive way of choosing the project to upgrade and a choice of which .NET version to target.

Different upgrade types

As mentioned earlier in the chapter, the .NET Upgrade Assistant supports three upgrade types. These upgrade types provide different upgrade experiences and depend on which project types are being upgraded.

In-place

This option upgrades your project all at once. Creating a separate branch for the upgraded project in your source control solution is prudent.

Side-by-side

The side-by-side option does not modify your original project. It places a copy of your project in your solution that contains the upgraded code. This is an excellent option if your project contains many dependencies that might break after the upgrade.

Side-by-side incremental

When dealing with web applications that require upgrading from ASP.NET to ASP.NET Core, this option is ideal because these two technologies are quite different. Using this upgrade type puts a .NET project next to your existing .NET Framework project. It routes endpoints implemented in the .NET project while routing all other calls to the .NET Framework application. The side-by-side incremental upgrade allows you to be in control and slowly upgrade your application bit by bit.

In conclusion

The .NET Upgrade Assistant supported C# and Visual Basic at the time of this writing. The supported projects are as follows:

- ASP.NET
- Azure Functions

- Windows Presentation Foundation
- Windows Forms
- Class libraries
- Console apps
- NET Native UWP
- Xamarin Forms
- .NET MAUI

The upgrade paths that are supported are:

- .NET Framework to .NET
- .NET Core to .NET
- UWP to WinUI 3
- Previous .NET version to the latest .NET version
- Azure Functions v1-v3 to v4 isolated
- Xamarin Forms to .NET MAUI
    - XAML file transformations only support upgrading namespaces. For more comprehensive transformations, use Visual Studio 2022 version 17.6 or later.

It goes without saying that you need to test your project after an upgrade. The .NET Upgrade Assistant is by no means a magic wand, but it allows for a much better developer experience when compared to upgrading your project manually.

For more information on the .NET Upgrade Assistant, you can browse this URL.

Chapter 2 A Closer Look at C# 11

C# 11 is a solid release when considering the new features it introduces. Some of these changes include:

- File-scoped types
- Generic math support
- UTF-8 literals
- Raw string literals
- Required members
- Generic attributes
- List patterns
- Pattern matching on spans
- Auto-default structs

Don’t let the standard term support status of .NET 7.0 fool you. The following sections show that .NET 7.0 offers exciting new features for developers.

Note: Throughout this book, I use Visual Studio 17.6.6.

Let’s have a look at some of these in more detail in the following sections.

File-scoped types

C# 11 introduces the **`file`** contextual keyword, which is a scope access modifier. This scopes the types to the file that they are in. This is very useful to source code generators because classes with the same name will not be an issue.

Consider the following class file called Student.cs. It contains the **`Student`** class as well as the **`Address`** class.

Code Listing 8: The Student Class and Address Class

```
internal class Student
{
    public string GetStudentAddress()
    {
        return new Address().ReturnAddressDetails();
    }
}

public class Address
{
    public string Address1 { get; set; }
    public string Address2 { get; set; }
    public string Address3 { get; set; }
    public string PostalCode { get; set; }

    public Address()
    {
        Address1 = "3012 William Nicol Drive";
        Address2 = "Bryanston";
        Address3 = "Johannesburg";
        PostalCode = "2191";
    }

    public string ReturnAddressDetails()
    {
        return $"{Address1}\n{Address2}\n{Address3}\n{PostalCode}";
    }
}
```

If I went ahead and added a class called **`Address`** to the solution, as seen in Figure 13, I would see an error.

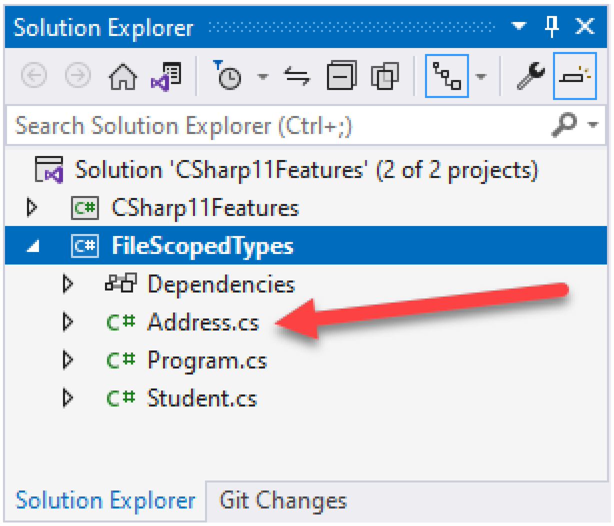
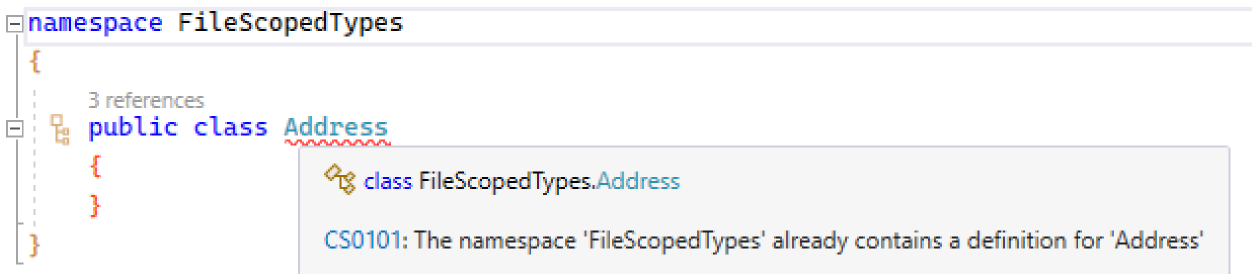


Figure 13: Adding a Class Called Address

Visual Studio correctly informs me that the newly added class is invalid because my project already contains a definition for the class **`Address`**.



```
namespace FileScopedTypes
{
    3 references
    public class Address
    {
    }
}
```

class FileScopedTypes.Address

CS0101: The namespace 'FileScopedTypes' already contains a definition for 'Address'

Figure 14: The Class Error

This is where file-scoped types shine. If I modify the code in Code Listing 8 and change the type modifier of the **`Address`** class from **`public`** to **`file`**, the **`Address`** class scope and visibility is restricted to the file it is created in.

Code Listing 9: Changing the Address Class from public to file

```
internal class Student
{
    public string GetStudentAddress()
    {
        return new Address().ReturnAddressDetails();
    }
}

file class Address
{
    public string Address1 { get; set; }
    public string Address2 { get; set; }
    public string Address3 { get; set; }
    public string PostalCode { get; set; }

    public Address()
    {
        Address1 = "3012 William Nicol Drive";
        Address2 = "Bryanston";
        Address3 = "Johannesburg";
        PostalCode = "2191";
    }

    public string ReturnAddressDetails()
    {
        return $"{Address1}\n{Address2}\n{Address3}\n{PostalCode}";
    }
}
```

Any types contained in the file-local type will also only be visible within the file in which it has been declared. And herein lies its usefulness.

Imagine that you wanted to create an extension method for your **Address** class. The functionality provided in the extension methods you write is specific to this class alone, and you do not want it to leak out to the rest of the project. Consider the code in Code Listing 10.

Code Listing 10: File-Scoped Extension Method Class

```
internal class Student
{
    public string GetStudentAddress()
    {
        return new Address().ReturnAddressDetails();
    }
}

file class Address
{
    public string Address1 { get; set; }
    public string Address2 { get; set; }
    public string Address3 { get; set; }
    public string PostalCode { get; set; }

    public Address()
    {
        Address1 = "3012 William Nicol Drive";
        Address2 = "Bryanston";
        Address3 = "Johannesburg";
        PostalCode = "2191";
    }

    public string ReturnAddressDetails()
    {
        return PostalCode.StringIsInt()
            ? $"{Address1}\n{Address2}\n{Address3}\n{PostalCode}"
            : throw new Exception("Invalid address");
    }
}

file static class ClassExtensions
{
    public static bool StringIsInt(this string value)
    {
        return int.TryParse(value, out _);
    }
}
```

The **ClassExtensions** class is created as **file static** so that it becomes scoped to the Student.cs file only.

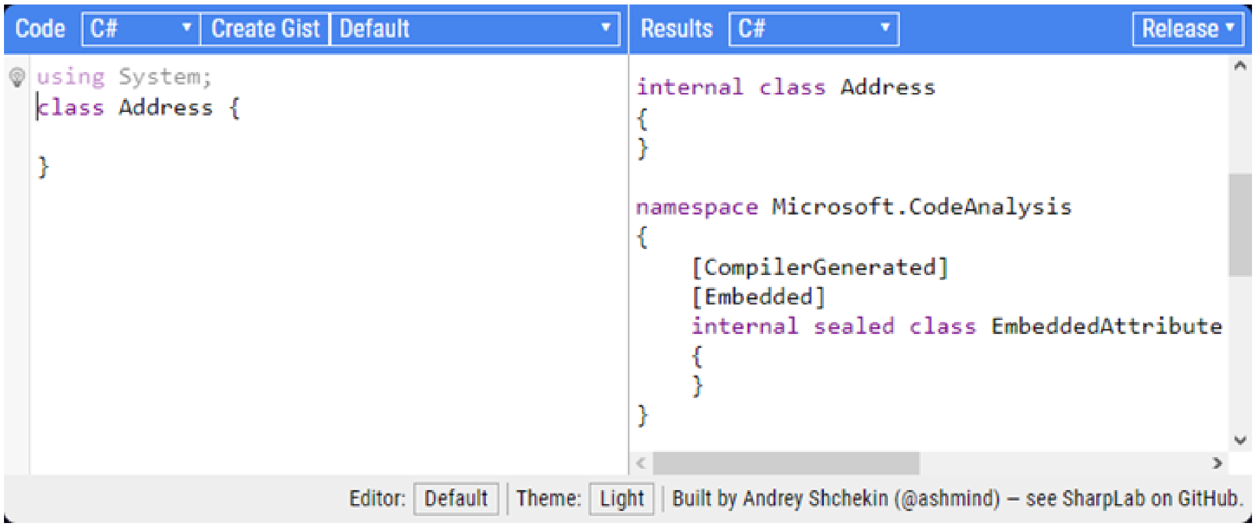
I believe that most developers will not use this feature too often. The main benefit here is when writing code generators. This is notoriously difficult to do without generating code that clashes with code already in the assembly. File-scoped types are, therefore, game-changing if you are writing source-code generators.

What the compiler sees

Why does this work? What does the compiler see when you create a file-scoped class? To illustrate this, I am using SharpLab, which allows you to see the code as the compiler sees it.

Note: SharpLab is a .NET code playground available at https://sharplab.io.

If we paste the code **`class Address { }`** into SharpLab, you see the compiled code it generates (Figure 15).

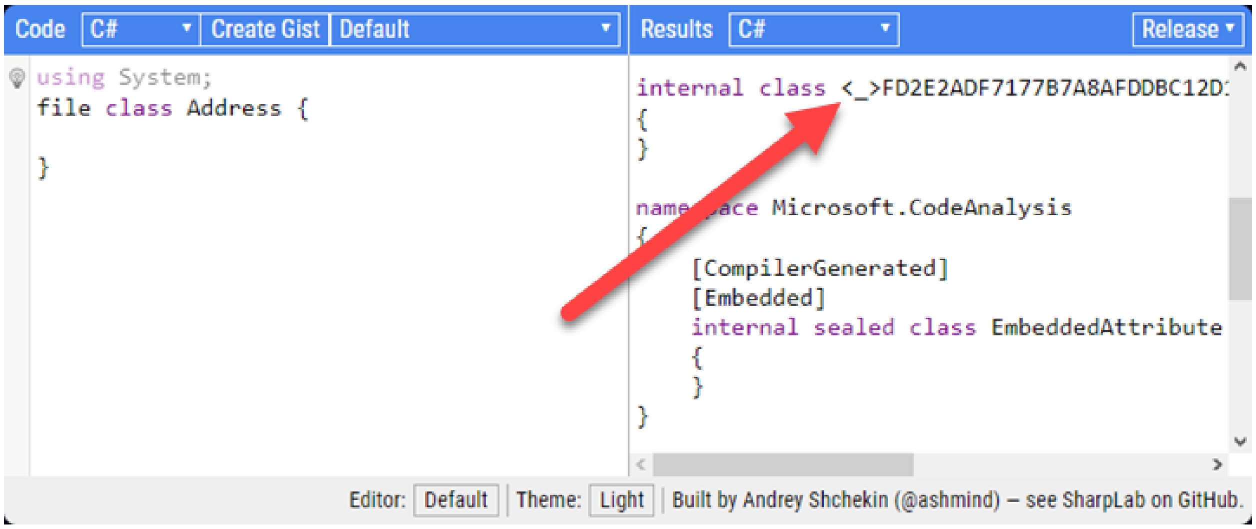


```
using System;
class Address {

}
```

```
internal class Address
{
}

namespace Microsoft.CodeAnalysis
{
    [CompilerGenerated]
    [Embedded]
    internal sealed class EmbeddedAttribute
    {
    }
}
```

Figure 15: SharpLab Compiled Code for Class Address

A class is internal by default, as illustrated in the compiled code. If we change **`class Address`** to **`file class Address`**, the compiled code changes.


Code C# Create Gist Default

```
using System;
file class Address {

}
```

Results C# Release

```
internal class <_>FD2E2ADF7177B7A8AFDDBC12D
{
}

namespace Microsoft.CodeAnalysis
{
    [CompilerGenerated]
    [Embedded]
    internal sealed class EmbeddedAttribute
    {
    }
}
```

Editor: Default | Theme: Light | Built by Andrey Shchekin (@ashmind) – see SharpLab on GitHub.

Figure 16: Compiled Code for File-Scoped Address Class

The screenshot in Figure 16 does not display the entire code, so I have included the code in Code Listing 11. Notice that the **`Address`** class is still an **`internal`** class but is compiled into a class with a name that is virtually impossible to replicate.

Code Listing 11: The Compiled Code

```
using System;
using System.Diagnostics;
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Security;
using System.Security.Permissions;
using Microsoft.CodeAnalysis;

[assembly: CompilationRelaxations(8)]
[assembly: RuntimeCompatibility(WrapNonExceptionThrows = true)]
[assembly:
Debuggable(DebuggableAttribute.DebuggingModes.IgnoreSymbolStoreSequencePoin
ts)]
[assembly: SecurityPermission(SecurityAction.RequestMinimum,
SkipVerification = true)]
[assembly: AssemblyVersion("0.0.0.0")]
[module: UnverifiableCode]
[module: System.Runtime.CompilerServices.RefSafetyRules(11)]

internal class
<_>FD2E2ADF7177B7A8AFDDBC12D1634CF23EA1A71020F6A1308070A16400FB68FDE__Addre
ss
{
}
```

```
namespace Microsoft.CodeAnalysis
{
    [CompilerGenerated]
    [Embedded]
    internal sealed class EmbeddedAttribute : Attribute
    {
    }
}

namespace System.Runtime.CompilerServices
{
    [CompilerGenerated]
    [Microsoft.CodeAnalysis.Embedded]
    [AttributeUsage(AttributeTargets.Module, AllowMultiple = false,
Inherited = false)]
    internal sealed class RefSafetyRulesAttribute : Attribute
    {
        public readonly int Version;

        public RefSafetyRulesAttribute(int P_0)
        {
            Version = P_0;
        }
    }
}
```

The compiled code **`internal class Address`** has changed to **`internal class <_>FD2E2ADF7177B7A8AFDDBC12D1634CF23EA1A71020F6A1308070A16400FB68FDE__Address`**

As a developer writing a source code generator, this allows me to use the class names that I want without passing assemblies around to check for conflicts first, resulting in faster source generation and simpler code.

Generic math support

New, math-related generic interfaces introduced to the base class library in .NET 7.0 means that you can constrain the type parameter of a generic method to be “number-like.” You can, therefore, perform mathematical operations generically.

To illustrate this concept, I am creating a **`Mathinator`** class that adds the values of an array together to produce a result.

Code Listing 12: The Mathinator Class

```
internal class Mathinator
{
    public int SumArray(int[] _values)
    {
        int result = 0;
        foreach (int i in _values)
        {
            result += i;
        }
        return result;
    }
}
```

In the consuming code, I create an instance of the **`Mathinator`** class and call the **`SumArray`** method.

Code Listing 13: Calling the SumArray Method

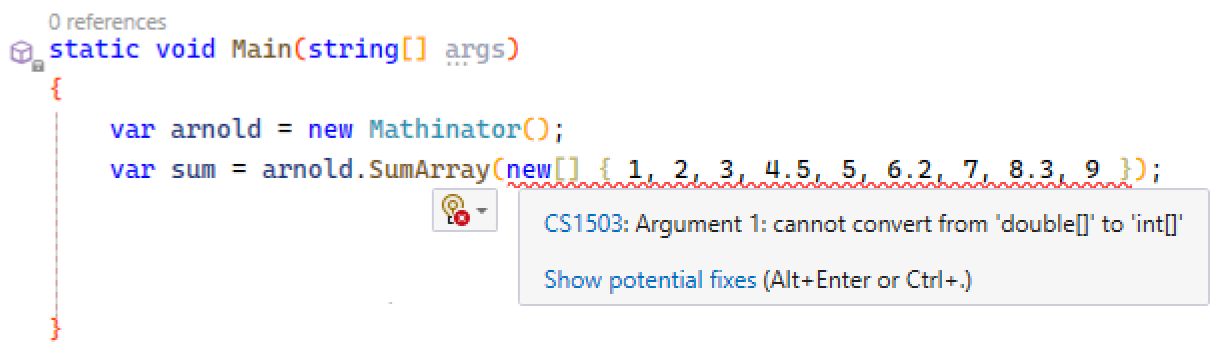
```
internal class Program
{
    static void Main(string[] args)
    {
        var arnold = new Mathinator();
        var sum = arnold.SumArray(new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 });
    }
}
```

Notice that the code expects an array of integers, and as long as the **`SumArray`** method receives an array of integers, the code works as expected.

Code Listing 14: Change the Array

```
internal class Program
{
    static void Main(string[] args)
    {
        var arnold = new Mathinator();
        var sum = arnold.SumArray(new[] { 1, 2, 3, 4.5, 5, 6.2, 7, 8.3, 9
});
    }
}
```

If, however, we change the array to contain doubles, as illustrated in Code Listing 14, the code does not compile.



```
0 references
static void Main(string[] args)
{
    var arnold = new Mathinator();
    var sum = arnold.SumArray(new[] { 1, 2, 3, 4.5, 5, 6.2, 7, 8.3, 9 });
}
```

CS1503: Argument 1: cannot convert from 'double[]' to 'int[]'

Show potential fixes (Alt+Enter or Ctrl+.)

Figure 17: The Argument Error

We can quickly fix this issue (Code Listing 15) by extending the **`Mathinator`** class to contain an overloaded method that handles arrays of doubles.

Code Listing 15: The Extended Mathinator Class

```
internal class Mathinator
{
    public int SumArray(int[] _values)
    {
        int result = 0;
        foreach (int i in _values)
        {
            result += i;
        }
        return result;
    }

    public double SumArray(double[] _values)
    {
        double result = 0.0;
        foreach (double i in _values)
        {
            result += i;
        }
        return result;
    }
}
```

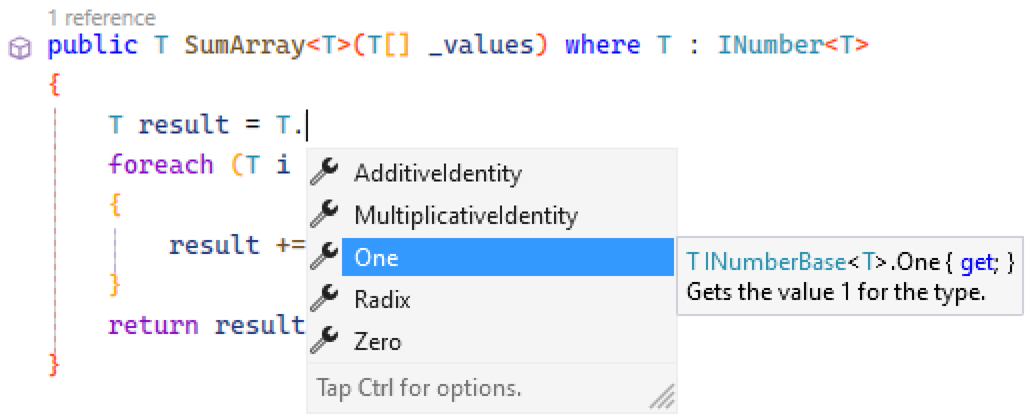
However, an alternative way to cater for this is by using generic math support. Consider the code illustrated in Code Listing 16. Notice that we now have a generic method with a constraint that specifies that **`T`** must be an **`INumber`** of **`T`**. Because **`INumber`** has the static member **`Zero`**, I can use **`T.Zero`** to provide a default value for the returned **`result`**.

Generic math support allows developers to be flexible when writing code dealing with numbers.

Code Listing 16: The Modified Mathinator Class

```
internal class Mathinator
{
    public T SumArray<T>(T[] _values) where T : INumber<T>
    {
        T result = T.Zero;
        foreach (T i in _values)
        {
            result += i;
        }
        return result;
    }
}
```

Looking closely at **`INumber`**, you notice that **`Zero`** is not the only property exposed.



```
1 reference
public T SumArray<T>(T[] _values) where T : INumber<T>
{
    T result = T.
    foreach (T i
    {
        result +=
    }
    return result
}
```

Figure 18: Properties of INumber

There are also several methods available for use. But why did this work? What has changed in C# 11 and .NET 7.0, and how does this generic math support open up more possibilities for developers? The answer is that C# 11 lets you define static virtual interface members. This new feature allows us to use generic algorithms to specify “number-like” behavior.

The [Microsoft documentation] defines **`static abstract`** and **`virtual`** members as follows:

“Beginning with C# 11, an interface may declare **`static abstract`** and **`static virtual`** members for all member types except fields. Interfaces can declare that implementing types must define operators or other static members. This feature enables generic algorithms to specify number-like behavior. You can see examples in the numeric types in the .NET runtime, such as **`System.Numerics.INumber<TSelf>`**.”

You might be wondering how this would apply to you in a real-world situation. I demonstrate this when I discuss minimal APIs in the next chapter. So, stick around; things are getting better from here on out.

UTF-8 literals

UTF-8 is considered the encoding of the web. It's also used in significant parts of the .NET stack. The network stack still makes use of constants in code. Think of **`HTTP/1.0\r\n`**, or **`AUTH`**, or even **`Contant-Length:`**, which we see so often.

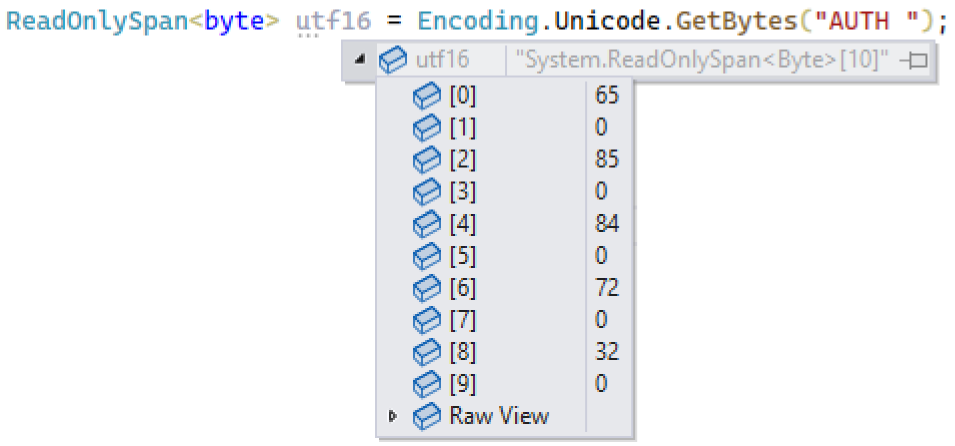
Unfortunately, there is no efficient syntax for dealing with these constants, and C# represents all strings as UTF-16 encoding.

With C# 11, you can mark strings as UTF-8 using the **`u8`** suffix, as seen in Code Listing 17.

Code Listing 17: Using the u8 Suffix on String Literals

```
ReadOnlySpan<byte> utf8 = "AUTH "u8;
```

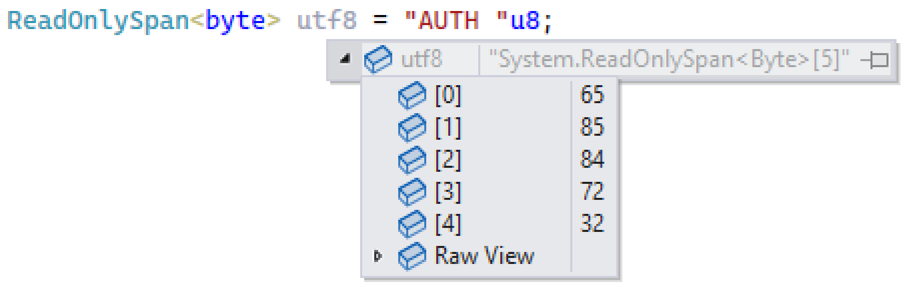
Compare the size difference of UTF-16 versus UTF-8.



```
ReadOnlySpan<byte> utf16 = Encoding.Unicode.GetBytes("AUTH ");
utf16    "System.ReadOnlySpan<Byte>[10]"
  [0]    65
  [1]    0
  [2]    85
  [3]    0
  [4]    84
  [5]    0
  [6]    72
  [7]    0
  [8]    32
  [9]    0
  Raw View
```

Figure 19: UTF-16 Encoding

As you can see when comparing Figure 19 and Figure 20, the **`utf16`** variable contains 10 bytes, while the **`uft8`** variable only contains 5 bytes.



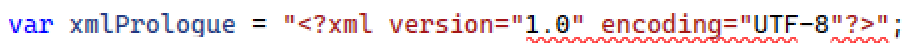
```
ReadOnlySpan<byte> utf8 = "AUTH "u8;
utf8    "System.ReadOnlySpan<Byte>[5]"
  [0]    65
  [1]    85
  [2]    84
  [3]    72
  [4]    32
  Raw View
```

Figure 20: UTF-8 Encoding

This difference is due to the size difference between the different encodings.

Raw string literals

The next feature makes life much easier for developers when working with strings. Consider the XML Prologue in Figure 21.



```
var xmlPrologue = "<?xml version="1.0" encoding="UTF-8"?>";
```

Figure 21: The Literal String Unescaped

Because the string is not escaped, the double quotes are invalid, because in C#, you cannot have a double quote inside a string. We generally overcome this by escaping the double quotes illustrated in Code Listing 18.

Code Listing 18: Escaping the Double Quotes

```
var xmlPrologue = "<?xml version=\"1.0\" encoding=\"UTF-8\"?>";
```

You can also escape the double quotes by using a verbatim string, but you still need to escape the double quotes by doubling them up.

Code Listing 19: Another Way to Escape Double Quotes

```
var xmlPrologue = @"<?xml version=""1.0"" encoding=""UTF-8""?>";
```

In C# 11, however, you can escape the double quotes inside the string by providing three double quotes around the string, as illustrated in Code Listing 20.

Code Listing 20: Raw String Literals

```
var xmlPrologue = """<?xml version="1.0" encoding="UTF - 8"?>""";
```

Raw string literals certainly help make the string more readable. If you, by chance, happen to have three double quotes in a string, then you would escape those by adding four double quotes around the string, as illustrated in Code Listing 21.

Code Listing 21: Escaping Three Double Quotes

```
var myString = """"This is a """ string with three double quotes"""";
```

This logic can continue indefinitely. In other words, if your string contains **n** double quotes, you have to start and end your string with **n+1** double quotes.

Raw string literals also work well when dealing with JSON strings. Consider the escaped JSON string in Code Listing 22.

Code Listing 22: Escaping a JSON String

```
var jsonString = "{\n    \"course\": \".NET 7 and C# 11 Succinctly\"\n}";
```

Running the program results in the JSON, as illustrated in Code Listing 23.

Code Listing 23: The JSON Output

```
{
    "course": ".NET 7 and C# 11 Succinctly"
}
```

The code in Code Listing 22 does not read very nicely and becomes more complicated when the JSON string becomes more complicated.

Code Listing 24: Working with JSON

```
var jsonString = """
{
    "course": ".NET 7 and C# 11 Succinctly"
}
""";
```

Using raw string literals, you can write a much better representation of your JSON string, as illustrated in Code Listing 24.

Using string interpolation

You might be wondering if string interpolation works with raw string literals.

Code Listing 25: Using String Interpolation

```
var foxColor = "brown";
var myString = $"""The quick {foxColor} fox""";
```

As illustrated in Code Listing 25, string interpolation works as expected. You need to do a little more work when working with strings that contain curly braces, such as in JSON.

Code Listing 26: String Interpolation with Strings Containing Curly Braces

```
var courseName = ".NET 7 and C# 11 Succinctly";
var jsonString = $$"""
{
    "course": "{{courseName}}"
}
""";
```

As illustrated in Code Listing 26, you must double up the **`$`** sign, and because the JSON string contains curly braces, you need to double up the curly braces around the **`courseName`** variable.

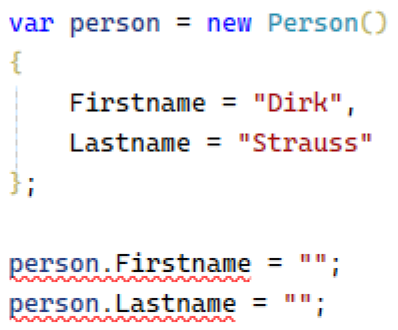
Required members

With C# 11, we finally have a way to resolve a particular bugbear of mine. Consider the **`Person`** class illustrated in Code Listing 27.

Code Listing 27: The Person Class

```
public class Person
{
    public string Firstname { get; init; }
    public string Lastname { get; init; }
}
```

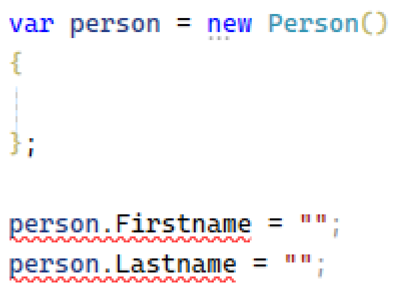
In C# 9, the introduction of the **`init`** keyword gave me a way only to allow the property to be set on the initialization of the class, as seen in Figure 22.



```
var person = new Person()
{
    Firstname = "Dirk",
    Lastname = "Strauss"
};

person.Firstname = "";
person.Lastname = "";
```

Figure 22: Using the init-Only Properties

Using **`init`** meant that if I wanted to change these properties later on, C# wouldn't allow me to do this. While this worked as intended, C# was all too happy to allow me not to specify a value for my **`init`** properties on the initialization of the class.



```
var person = new Person()
{

};

person.Firstname = "";
person.Lastname = "";
```

Figure 23: Not Providing Property Values on Initialization

You can see this behavior in Figure 23. While it tells me that I can only assign my **`init`**-only properties in an object initializer, it does not have a problem with the fact that I never assigned any values to begin with. If I wanted to enforce that these properties be set on initialization, I would have to demand it from the constructor, as illustrated in Code Listing 28.

Code Listing 28: Forcing Values to be Set

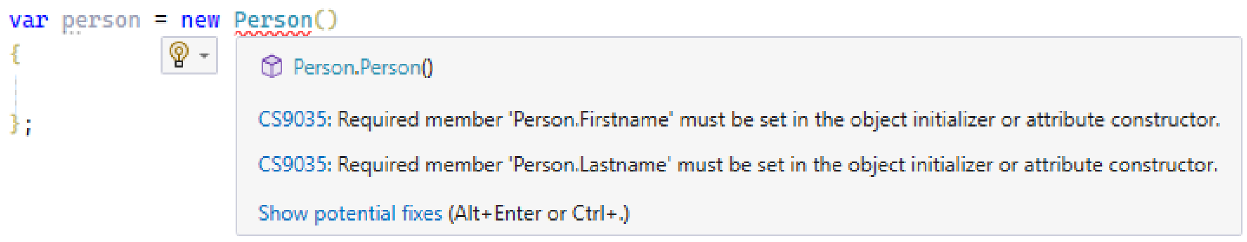
```
public class Person
{
    public string Firstname { get; init; }
    public string Lastname { get; init; }

    public Person(string firstName, string lastName)
    {
        Firstname = firstName;
        Lastname = lastName;
    }
}
```

Required members in C# 11 provide a more elegant approach to this issue.

Code Listing 29: Using the Required Keyword

```
public class Person
{
    public required string Firstname { get; init; }
    public required string Lastname { get; init; }
}
```

As illustrated in Code Listing 29, I can now tell C# that these properties must be assigned when initializing the class.



```
var person = new Person()
{

};
```

Person.Person()

CS9035: Required member 'Person.Firstname' must be set in the object initializer or attribute constructor.

CS9035: Required member 'Person.Lastname' must be set in the object initializer or attribute constructor.

Show potential fixes (Alt+Enter or Ctrl+.)

Figure 24: Compiler Error on Unset Required Properties

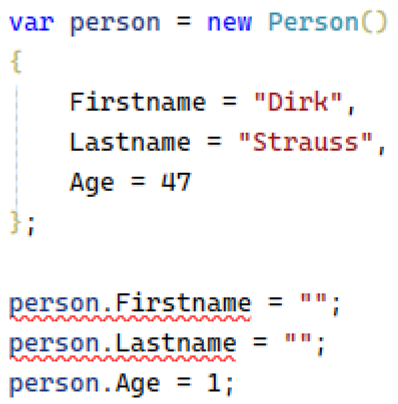
If I do not set the values on initialization, C# 11 complains and tells me that I need to provide values for these properties. It's worth noting that the **`required`** keyword is not specific to the **`init`** keyword. It can easily be used with a getter and setter, as illustrated in Code Listing 30.

Code Listing 30: Extending the Person Class

```
public class Person
```

```
{
    public required string Firstname { get; init; }
    public required string Lastname { get; init; }
    public required int Age { get; set; }
}
```

This means that, while I can only set **`Firstname`** and **`Lastname`** on object initialization, all three properties are required, allowing me to set a different value for the **`Age`** property after object initialization.



```
var person = new Person()
{
    Firstname = "Dirk",
    Lastname = "Strauss",
    Age = 47
};

person.Firstname = "";
person.Lastname = "";
person.Age = 1;
```

Figure 25: Setting Age after Initialization

Required members give me more control over the classes I create, allowing me to express my intent more clearly to consuming code.

Generic attributes

Let me start by defining what an attribute is. According to the Microsoft documentation:

Information provided by an attribute is also known as metadata. Metadata can be examined at run time by your application to control how your program processes data, or before run time by external tools to control how your application itself is processed or maintained.

I have created a filter for the boilerplate weather forecast API project in Visual Studio to demonstrate generic attributes. The filter allows me to do something before and after an endpoint call.

Code Listing 31: My Log Filter

```
public class LogFilter : IAsyncActionFilter
{
    public ILogger<LogFilter> _logger;

    public LogFilter(ILogger<LogFilter> logger) => _logger = logger;
    public async Task OnActionExecutionAsync(
        ActionExecutingContext context,
        ActionExecutionDelegate next)
    {
        _logger.LogInformation("Action started");
        await next();
        _logger.LogInformation("Action ended");
    }
}
```

This filter is illustrated in Code Listing 31. To use my filter, I need to add it to the service collection in the **`Program`** class by adding the code **`builder.Services.AddSingleton<LogFilter>();`**.

I can now apply this to a **`ServiceFilter`** attribute on the **`GET`** endpoint of my **`WeatherForecast`** API endpoint, as illustrated in Code Listing 32.

Code Listing 32: The GET Endpoint

```
[HttpGet(Name = "GetWeatherForecast")]
[ServiceFilter(typeof(LogFilter))]
public IEnumerable<WeatherForecast> Get()
{
    return Enumerable.Range(1, 5).Select(index => new WeatherForecast
    {
        Date = DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
        TemperatureC = Random.Shared.Next(-20, 55),
        Summary = Summaries[Random.Shared.Next(Summaries.Length)]
    })
    .ToArray();
}
```

You can see the information log in the output window by running the API and calling the **`GetWeatherForecast`** endpoint from Swagger or Postman.

But here is the issue with the **`ServiceFilter`** attribute, in that it takes a type as a parameter in the constructor, which is passed as **`typeof(LogFilter)`**. You can see this by looking at the **`ServiceFilterAttribute`** class illustrated in Code Listing 33.

Code Listing 33: The ServiceFilterAttribute Code

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method,
AllowMultiple = true, Inherited = true)]
    [DebuggerDisplay("ServiceFilter: Type={ServiceType} Order={Order}")]
    public class ServiceFilterAttribute : Attribute, IFilterFactory,
IOrderedFilter
    {
        public ServiceFilterAttribute(Type type)
        {
            ServiceType = type ?? throw new
ArgumentNullException(nameof(type));
        }

        public int Order { get; set; }
        public Type ServiceType { get; }
        public bool IsReusable { get; set; }


        public IFilterMetadata CreateInstance(IServiceProvider
serviceProvider)
        {
            if (serviceProvider == null)
            {
                throw new ArgumentNullException(nameof(serviceProvider));
            }

            var filter =
(IFilterMetadata)serviceProvider.GetRequiredService(ServiceType);
            if (filter is IFilterFactory filterFactory)
            {
                // Unwrap filter factories.
                filter = filterFactory.CreateInstance(serviceProvider);
            }

            return filter;
        }
    }
```

Let’s create our own service filter attribute as a generic attribute.

Code Listing 34: Our Own ServiceFilterAttribute

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method,
AllowMultiple = true, Inherited = true)]
public class ServiceFilterGenericAttribute<T> : Attribute,
IFilterFactory, IOrderedFilter
{
    public int Order { get; set; }
    public bool IsReusable { get; set; }

    public IFilterMetadata CreateInstance(IServiceProvider
serviceProvider)
    {
        if (serviceProvider == null)
        {
            throw new ArgumentNullException(nameof(serviceProvider));
        }

        var filter =
(IFilterMetadata)serviceProvider.GetRequiredService(typeof(T));
        if (filter is IFilterFactory filterFactory)
        {
            // Unwrap filter factories.
            filter = filterFactory.CreateInstance(serviceProvider);
        }

        return filter;
    }
}
```

Code Listing 34 illustrates that we no longer use the type passed in the constructor but instead use **`ServiceFilterGenericAttribute<T>`** and **`typeof(T)`** to set the filter.

I can modify my **`GetWeatherForecast`** endpoint as illustrated in Code Listing 35—not to use type parameters, but rather to use **`[ServiceFilterGeneric<LogFilter>]`**. Generic attributes, therefore, mean that my attribute class is generic.

Code Listing 35: The Modified GET Endpoint

```
[HttpGet(Name = "GetWeatherForecast")]
[ServiceFilterGeneric<LogFilter>]
public IEnumerable<WeatherForecast> Get()
{
    return Enumerable.Range(1, 5).Select(index => new WeatherForecast
    {
        Date = DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
        TemperatureC = Random.Shared.Next(-20, 55),
        Summary = Summaries[Random.Shared.Next(Summaries.Length)]
    })
    .ToArray();
}
```

The functionality here has remained the same, but in terms of writing better code, using generic attributes has made a significant improvement.

List patterns

C# 11 allows developers to match an array or list against a sequence of patterns. It means that list patterns apply to anything that is countable or has a count property, for example, an array or list.

Code Listing 36: List Patterns on an Integer Array

```
int[] nums = { 1, 2, 3 };

Console.WriteLine(nums is [1, 2, 3]); // true
Console.WriteLine(nums is [1, 7, 3]); // false
Console.WriteLine(nums is [1, 2, 3, 4]); // false
Console.WriteLine(nums is [0 or 5]); // false
```

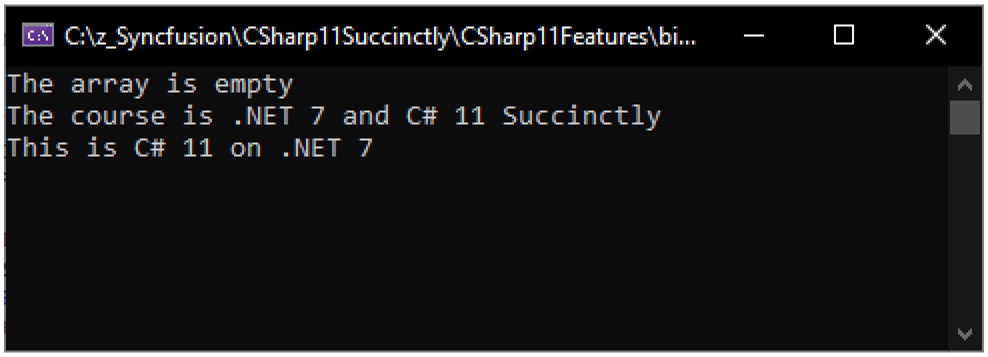
The application of this means that, considering the code illustrated in Code Listing 36, the matched list patterns return **`true`**, and if no match is found, **`false`**.

But we can go even further when we consider the code in Code Listing 37.

Code Listing 37: Matching String Arrays

```
var empty = Array.Empty<string>();
// outputMatch1 will output "The array is empty"
var outputMatch1 = empty switch
{
    [] => "The array is empty",
    [var fullCourse] => $"The course is {fullCourse}",
    [var netFx, var lang] => $"This is {lang} on {netFx}",
    _ => "No patterns matched"
};

var course = new[] { ".NET 7 and C# 11 Succinctly" };
// outputMatch2 will output "The course is .NET 7 and C# 11 Succinctly"
var outputMatch2 = course switch
{
    [] => "The array is empty",
    [var fullCourse] => $"The course is {fullCourse}",
    [var netFx, var lang] => $"This is {lang} on {netFx}",
    _ => "No patterns matched"
};

var dotNetAndCSharp = new[] { ".NET 7", "C# 11" };
// outputMatch3 will output "This is C# 11 on .NET 7"
var outputMatch3 = dotNetAndCSharp switch
{
    [] => "The array is empty",
    [var fullCourse] => $"The course is {fullCourse}",
    [var netFx, var lang] => $"This is {lang} on {netFx}",
    _ => "No patterns matched"
};

Console.WriteLine(outputMatch1);
Console.WriteLine(outputMatch2);
Console.WriteLine(outputMatch3);
```

Before each **switch** expression, we declare a string array and then pass it to the **switch** to be matched.



```
The array is empty
The course is .NET 7 and C# 11 Succinctly
This is C# 11 on .NET 7
```

Figure 26: The Matched List Patterns

Running the application produces the result in Figure 26. I can't say much else about list patterns other than they provide another flexible way to match things. The use cases might be limited, but they could come in handy somewhere down the line.

Pattern matching on spans

Pattern matching has allowed developers to test whether a string has a particular constant. This same pattern-matching logic now comes to **Span<char>** or **ReadOnlySpan<char>** variables.

Code Listing 38: Pattern Matching on a ReadOnlySpan<T>

```
ReadOnlySpan<char> value = ".NET 7 and C# 11 Succinctly";

if (value is ['.', ..])
{
    Console.WriteLine("The string starts with a dot");
}
```

Consider the code in Code Listing 38. We have a variable of type **ReadOnlySpan<char>**, and we can now use pattern matching on this variable to test if the string starts with a period.

Auto-default struct

C# 11 now ensures that all fields of a **struct** type are initialized to their respective default values. This is automatically done by the compiler. This enhancement is best illustrated using SharpLab, which you can find at sharplab.io.

Consider the code in Code Listing 39.

Code Listing 39: The Point struct

```
public struct Point
{
    public int X;
    public int Y;

    public Point(int x, int y)
    {
        X = x;
        //Y = y;
    }
}
```

The fields **X** and **Y** are initialized inside the constructor of the **struct**. In the previous version of C#, however, if you comment out the initialization of one of the fields (**Y** in this example), you receive an error stating that the field **Y** needs to be assigned.

In C# 11, this has changed, and you can see why when you look at what the compiler does.

Code Listing 40: The Compiler Auto-Defaults the struct

```
public struct Point
{
    public int X;
    public int Y;

    public Point(int x, int y)
    {
        Y = 0;
        X = x;
    }
}
```

Viewing the compiler-generated code in Code Listing 40, the field **Y** is automatically initialized to its default value.

In conclusion

.NET 7 and C# 11 introduce developers to many great new features. As we will see in the next chapter, ASP.NET Core 7 also brings many improvements and features that enhance developer productivity and the efficiency of their code.

Chapter 3 ASP.NET Core 7

Chances are, you will work with ASP.NET Core at some point in your career. Whether this is developing a website or an API, ASP.NET Core is at play here. Due to the importance of ASP.NET Core, Microsoft has added a large number of features to the ASP.NET Core platform. Let's look at some of these through the lens of .NET 7.

How does ASP.NET Core fit into the whole .NET ecosystem?

When we talk about ASP.NET Core, we talk about a methodology. In other words, we are referring to a few things combined. As illustrated in Figure 27, ASP.NET Core comprises a core platform and the specific technology you use for your application.

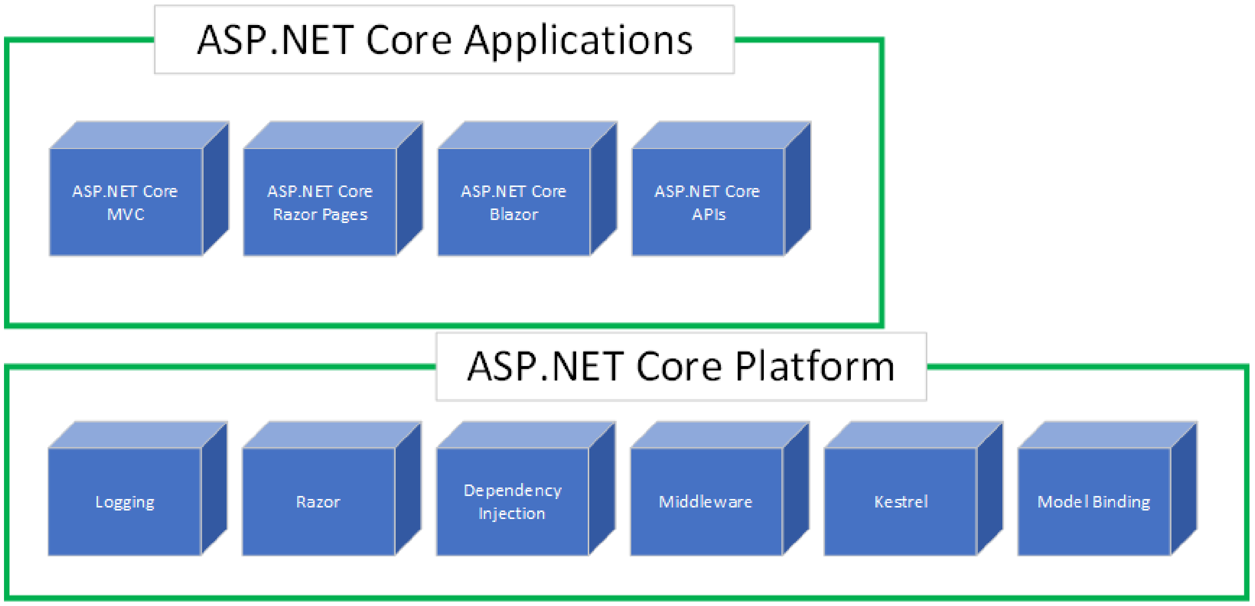


Figure 27: ASP.NET Core

The components in the core platform, such as middleware, logging, Razor pages, etc., make up the building blocks of this shared platform. On top of this core platform sit various frameworks developers can use to create applications.

This, in essence, is what ASP.NET Core looks like. You can't use one of the application frameworks without one or more core platform components. Of course, the illustration in Figure 27 oversimplifies ASP.NET Core to a large extent. It is just a high-level overview of what ASP.NET Core is.

Let's have a look at some of the new features of ASP.NET Core 7.

New features in ASP.NET Core 7

In this section, we'll look at several new features in ASP.NET Core 7 while discussing a few in detail later in the chapter.

HTTP/2 WebSocket support and performance improvements

.NET 7 introduces WebSockets over HTTP/2 support for Blazor WebAssembly, the JavaScript client, and the Kestrel web server. Being able to use WebSockets over HTTP/2 allows developers to take advantage of new features such as:

- Header compression.
- Multiplexing, reducing the time and resources across multiple server requests.

.NET 7 introduced a significant re-architecture of how Kestrel processes HTTP/2 requests. These performance improvements result in ASP.NET Core applications with busy HTTP/2 connections experiencing reduced CPU usage while enjoying a higher throughput.

HTTP/3 improvements

ASP.NET Core brings improvements to HTTP/3, specifically:

- HTTP/3 is no longer experimental and is fully supported in ASP.NET Core.
- Kestrel support for HTTP/3 is improved, including performance and feature parity with HTTP/1.1 and HTTP/2.
- Full support for **UseHttps(ListenOptions, X509Certificate2)** is provided with HTTP/3.

HTTP/3 support has been added on HTTP.sys and IIS.

Tip: HTTP.sys is an alternative web server to the Kestrel server. It offers some features not available with Kestrel. HTTP.sys is a web server for ASP.NET Core that only runs on Windows.

.NET 7 also reduces HTTP/3 allocations. If you are interested in seeing some of these improvements, log in to GitHub, navigate to https://github.com/dotnet/aspnetcore, and then search the repository issues for the following pull items:

- 42685
- 42708
- 42760

Output caching

Output caching stores responses from your web application and allows ASP.NET Core to serve them from a cache, instead of computing them each time. It differs from response caching in the following ways:

- You can configure the caching behavior on the server.
- You can programmatically invalidate cache entries.
- The risks of cache stampede and thundering herd are mitigated by resource locking.
- The server can return a **`304 Not Modified`** HTTP status instead of a cached response body.
- The cache storage medium is extensible.

Output caching is new middleware in ASP.NET Core 7.

Rate limiting

ASP.NET Core 7 applications can now configure rate-limiting policies and attach those policies to endpoints in your application.

Request decompression

Request decompression is also new middleware in ASP.NET Core 7. It allows API endpoints to accept requests with compressed content. Using the Content-Encoding HTTP header, compressed request content can be automatically identified and decompressed. This means that developers do not need to write custom code to handle compressed requests.

Improvements to hot reload

Hot reload in .NET 6 significantly improved the developer workflow when debugging applications. Referring to the Hot Reload section in this article, in .NET 7, hot reload will be present for Blazor WebAssembly, allowing developers to:

- Add new types.
- Add nested classes.
- Add static and instance methods to existing types.
- Add static fields and methods to existing types.
- Add static lambdas to existing types.
- Add lambdas that capture **`this`** to existing methods that already captured **`this`** previously.

Rate limiting

There might be situations where applications come under extremely heavy load. This increases the possibility that the application would simply go down because it can't handle the increased load. The increase in load could be accidental or on purpose. An accidental load increase could result from a faulty script that repeatedly makes requests to an API endpoint. On the other hand, a malicious actor could willfully make multiple requests to your application to bring down the site.

Rate limiting in ASP.NET Core 7 allows developers to mitigate these issues:

- Your application can limit the number of requests in a given time.

- Any requests above the limit will not be handled.
- Developers can specify rate limiting globally or on a per-user basis.

Developers can also create combinations, such as rate limiting the application to a thousand requests per minute, but only a hundred requests per minute, per user.

Rate limiting allows developers to protect the application out of the box without adding any additional services. This prevents accidental or intentional abuse of the web application or API. Rate limiting comes in the form of middleware and works through policies attached to endpoints. This allows developers to specify different policies for different endpoints, which in turn allows the implementation of different limits to each endpoint.

ASP.NET Core 7 comes with several built-in rate limiter types. These are:

- Fixed window limit: For example, 10 requests per minute.
- Sliding window limit: Allowing a certain number of requests per segment of the window.
- Concurrency limit: Specifying how many concurrent requests are allowed.
- Token bucket limit: A more complex version of the sliding window limit that allows bursts of requests.

As mentioned earlier, any requests that exceed your limit will be rejected. It is, however, possible to set up rate limiting to queue some requests that go above the **`PermitLimit`** you set by using the **`QueueLimit`** option. This, in essence, delays the requests instead of rejecting them immediately.

Using a fixed window limit

Let's look at a code example implementing rate limiting. For this example, illustrated in Code Listing 41, I have just used the boilerplate code for the Weather Forecast API in Visual Studio. I am not using top-level statements or minimal APIs.

Tip: You can find the full source code for this book on GitHub.

I create a constant string for the rate limit policy name in the **`Program`** class. Here, I simply called it **`fixed`**.

I then add rate limiting to the services collection by:

- Specifying **`builder.Services.AddRateLimiter`**.
- Adding a fixed window limiter, **`AddFixedWindowLimiter`**, specifying the policy name.
- Providing options for the **`Window`** and **`PermitLimit`** options.

In this rate limiter, I will allow only four requests to be made in a time span of 20 seconds.

I then need to enable the rate limiting middleware by adding **`app.UseRateLimiter()`**, and I specify that I require rate limiting on my endpoints with **`app.MapControllers().RequireRateLimiting(FIXED_RATE_LIMIT_POLICY)`**, passing in the name of the policy I want to apply.

I am using **`MapControllers`** here, but I could also be more specific if I wanted.

Code Listing 41: Adding a Fixed Window Limiter

```
public class Program
{
    const string FIXED_RATE_LIMIT_POLICY = "fixed";

    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);

        builder.Services.AddControllers();
        builder.Services.AddEndpointsApiExplorer();
        builder.Services.AddSwaggerGen();

        // Fixed rate limit.
        builder.Services.AddRateLimiter(_ => _
        .AddFixedWindowLimiter(policyName: FIXED_RATE_LIMIT_POLICY,
options =>
        {
            options.PermitLimit = 4;
            options.Window = TimeSpan.FromSeconds(20);
        }));

        var app = builder.Build();

        if (app.Environment.IsDevelopment())
        {
            app.UseSwagger();
            app.UseSwaggerUI();
        }

        // Enable rate limiting middleware.
        app.UseRateLimiter();
        app.UseHttpsRedirection();
        app.UseAuthorization();

app.MapControllers().RequireRateLimiting(FIXED_RATE_LIMIT_POLICY);
        app.Run();
    }
}
```

Running the application in Swagger, I can execute the **`GET`** request for the **`/WeatherForecast`** API endpoint, as seen in Figure 28.

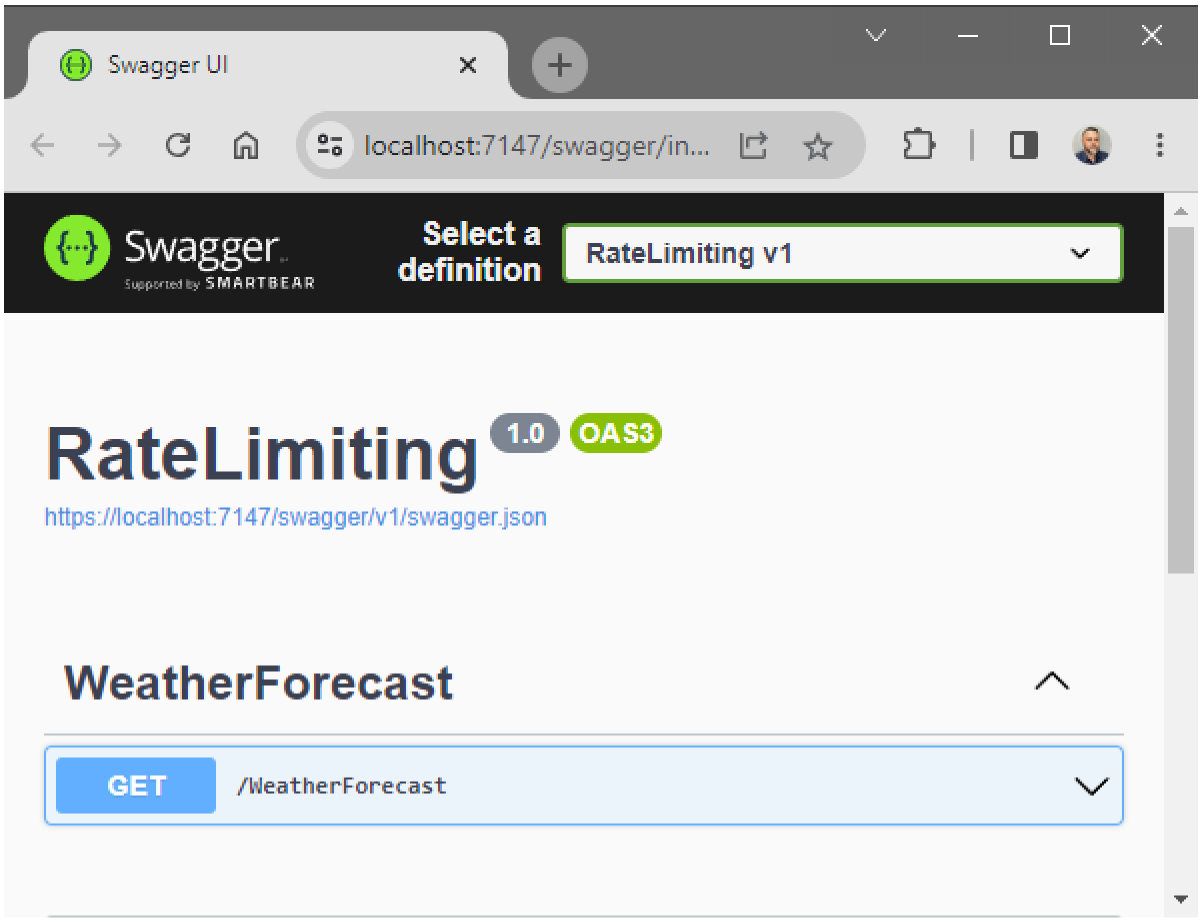
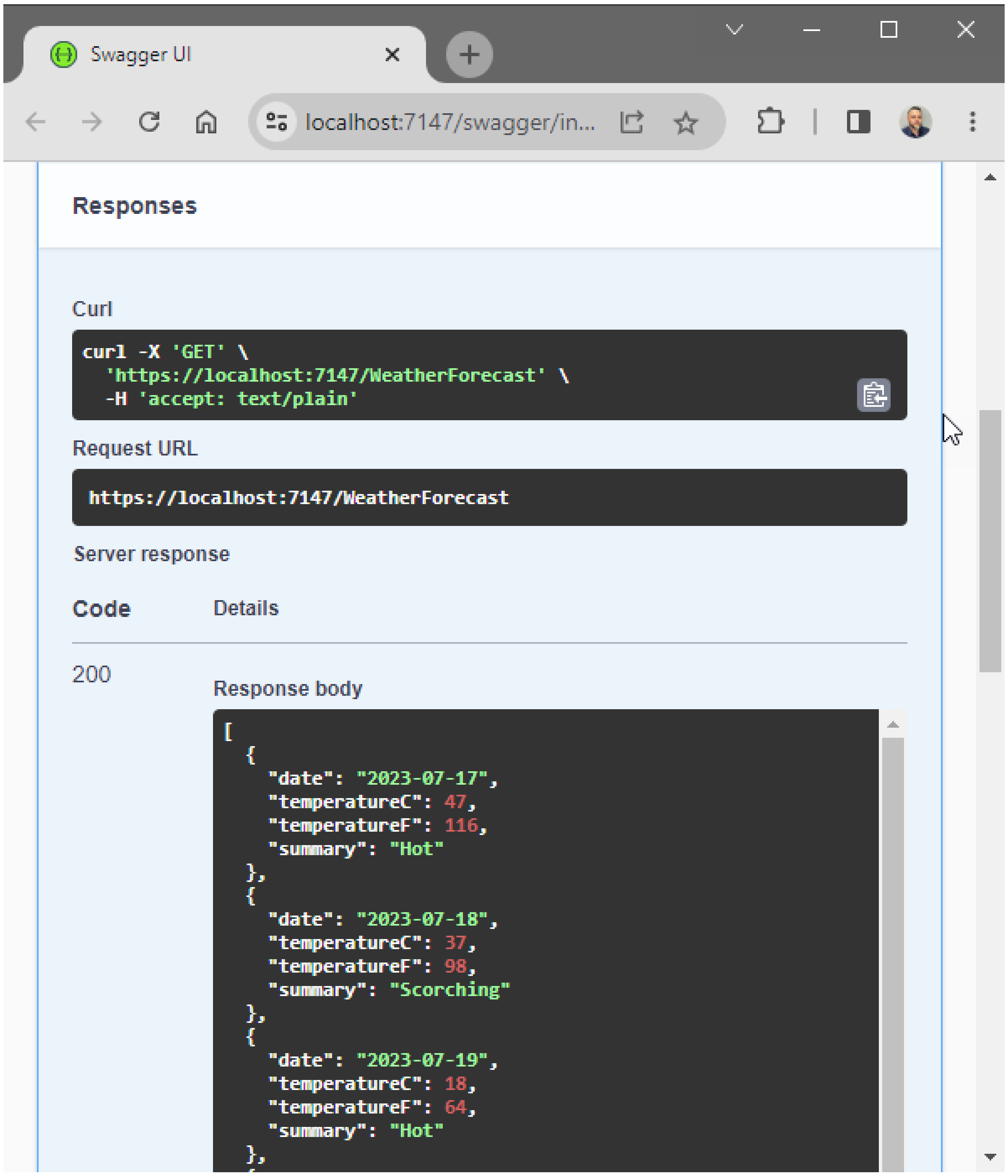


Figure 28: The GET Request in Swagger

When sending a **`GET`** request via Swagger to the **`WeatherForecast`** API, I receive the response shown in Figure 29.



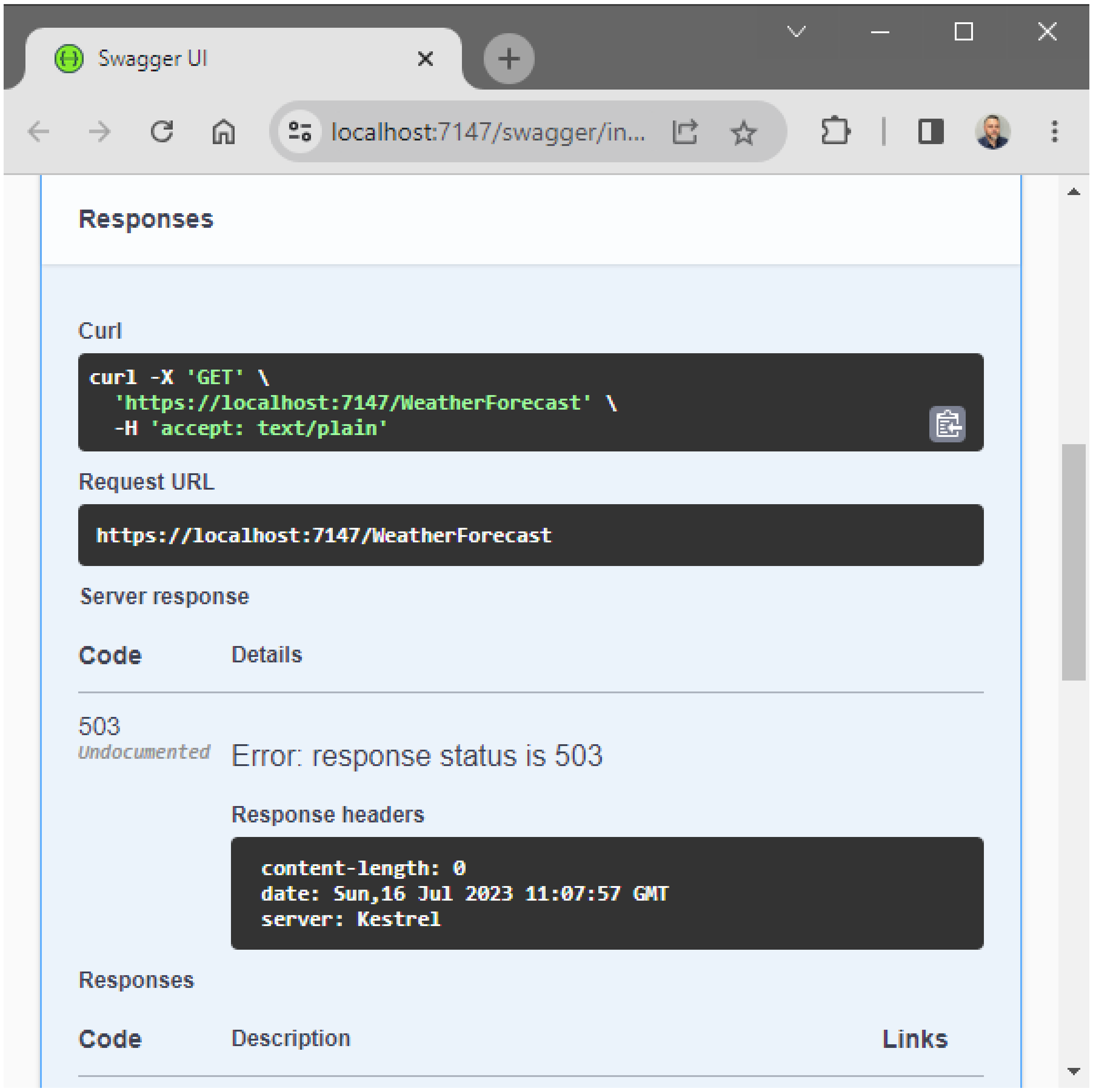
```
curl -X 'GET' \
  'https://localhost:7147/WeatherForecast' \
  -H 'accept: text/plain'
```

```
https://localhost:7147/WeatherForecast
```

```
[
  {
    "date": "2023-07-17",
    "temperatureC": 47,
    "temperatureF": 116,
    "summary": "Hot"
  },
  {
    "date": "2023-07-18",
    "temperatureC": 37,
    "temperatureF": 98,
    "summary": "Scorching"
  },
  {
    "date": "2023-07-19",
    "temperatureC": 18,
    "temperatureF": 64,
    "summary": "Hot"
  },
```

Figure 29: The GET Request Response

The response from the **`GET`** request is what we would expect. To hit the rate limit set by the **`PermitLimit`** option (which we specified as a value of **`4`**), we will receive a **`503`** response status on the fifth request, as seen in Figure 30.



Responses

Curl

```
curl -X 'GET' \
  'https://localhost:7147/WeatherForecast' \
  -H 'accept: text/plain'
```

Request URL

```
https://localhost:7147/WeatherForecast
```

Server response

Code	Details
503 *Undocumented*	Error: response status is 503

Response headers

```
content-length: 0
date: Sun,16 Jul 2023 11:07:57 GMT
server: Kestrel
```

Responses

Code	Description	Links

Figure 30: The Fifth GET Request is Rate Limited

This is because we performed more than four requests in the specified 20-second window. When the 20-second window has passed, we will receive a response again when performing the same **`GET`** request.

Enabling the queue

You will remember that it is possible to set up rate limiting to queue some requests that go above the **`PermitLimit`** you set by using the **`QueueLimit`** option. This, in essence, delays the requests instead of rejecting them immediately. To implement this, we need to modify our rate limiter slightly, as illustrated in Code Listing 42.

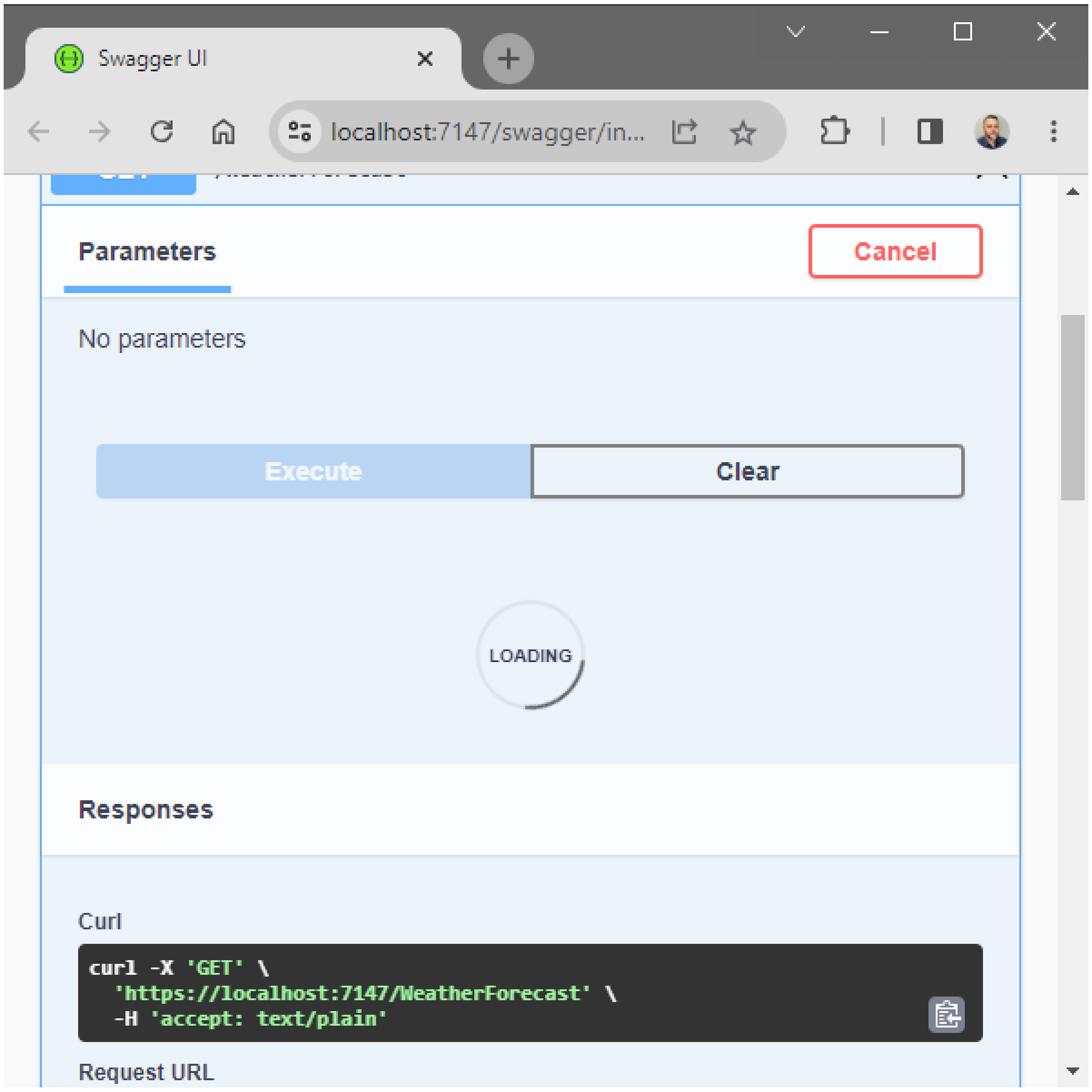
Code Listing 42: Enabling the Queue Limit

```
public class Program
{
    const string FIXED_RATE_LIMIT_POLICY = "fixed";

    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);

        builder.Services.AddControllers();
        builder.Services.AddEndpointsApiExplorer();
        builder.Services.AddSwaggerGen();

        // Fixed rate limit.
        builder.Services.AddRateLimiter(_ => _
        .AddFixedWindowLimiter(policyName: FIXED_RATE_LIMIT_POLICY,
options =>
        {
            options.PermitLimit = 4;
            options.Window = TimeSpan.FromSeconds(20);
            options.QueueProcessingOrder =
QueueProcessingOrder.OldestFirst;
            options.QueueLimit = 2;
        }));

        var app = builder.Build();

        if (app.Environment.IsDevelopment())
        {
            app.UseSwagger();
            app.UseSwaggerUI();
        }

        // Enable rate limiting middleware.
        app.UseRateLimiter();
        app.UseHttpsRedirection();
        app.UseAuthorization();

app.MapControllers().RequireRateLimiting(FIXED_RATE_LIMIT_POLICY);
        app.Run();
    }
}
```

For this to work, add the statement **`using System.Threading.RateLimiting`**, and add **`options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst`** and **`options.QueueLimit = 2`** to your rate limiter.

This will queue two requests, after which my rate limiter will process them from oldest to newest.

Running your API with the **`QueueLimit`** enabled and performing five **`GET`** requests will cause the API call to wait instead of returning a **`503`** response.



```
curl -X 'GET' \
  'https://localhost:7147/WeatherForecast' \
  -H 'accept: text/plain'
```

Figure 31: GET Request Queued

After the time has passed, the API will return a response. If I had to open three browser windows, hit the rate limit on the first browser that queues the fifth request, and perform a new **GET** request in the second browser, the **QueueLimit** of **2** will be full. If I perform a new **GET** request in the third browser, that request will return a **503**.

This is because my queue limit has been reached, and any further requests will be rejected.

Concurrency limit

Another option is to use the concurrency limiter, as illustrated in Code Listing 43.

Code Listing 43: Using the Concurrency Limiter

```
public class Program
{
    const string CONCURRENT_RATE_LIMIT_POLICY = "concurrent";

    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);

        builder.Services.AddControllers();
        builder.Services.AddEndpointsApiExplorer();
        builder.Services.AddSwaggerGen();

        // Concurrent rate limit.
        builder.Services.AddRateLimiter(_ => _
        .AddConcurrencyLimiter(policyName: CONCURRENT_RATE_LIMIT_POLICY,
options =>
        {
            options.PermitLimit = 4;
            options.QueueProcessingOrder =
QueueProcessingOrder.OldestFirst;
            options.QueueLimit = 2;
        }));

        var app = builder.Build();

        if (app.Environment.IsDevelopment())
        {
            app.UseSwagger();
            app.UseSwaggerUI();
        }

        // Enable rate limiting middleware.
        app.UseRateLimiter();
        app.UseHttpsRedirection();
        app.UseAuthorization();
app.MapControllers().RequireRateLimiting(CONCURRENT_RATE_LIMIT_POLICY);
        app.Run();
    }
}
```

This will limit the number of concurrent requests. As you can see, it is very similar to the fixed rate limiter.

I add the concurrent rate limiter to the services collection by:

- Specifying **`builder.Services.AddRateLimiter`**.
- Adding a concurrency limiter, **`AddConcurrencyLimiter`**, specifying the policy name.
- Specifying through the options that I only want to allow four concurrent requests to happen at the same time.

To apply this rate limiter, I add **`app.MapControllers().RequireRateLimiting(CONCURRENT_RATE_LIMIT_POLICY)`** to the **`MapControllers()`** method.

Global rate limiting

As the code in Code Listing 44 illustrates, applying a global rate limiter is possible.

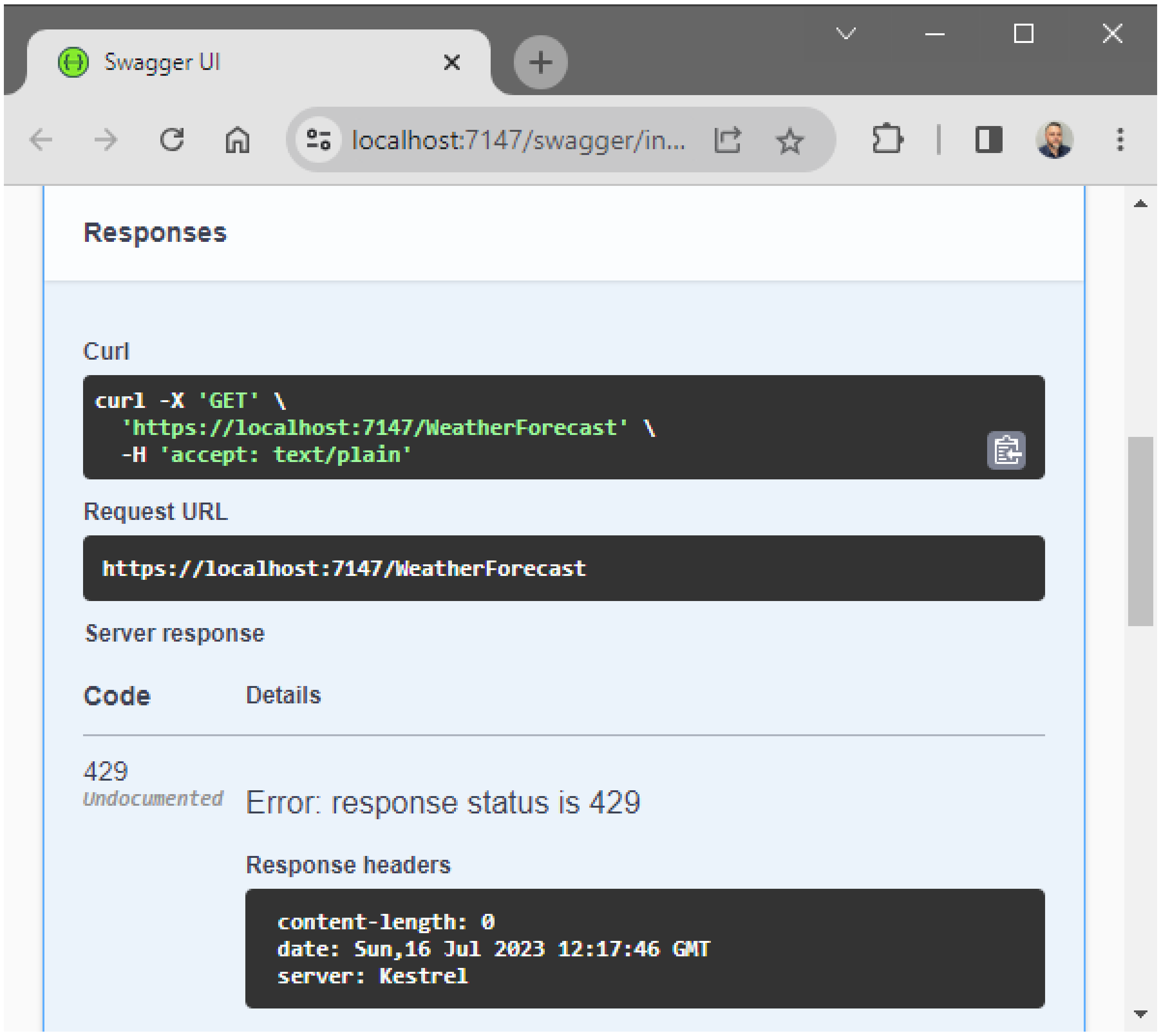
Code Listing 44: Adding Global Rate Limiting

```
public class Program
{
    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);

        builder.Services.AddControllers();
        builder.Services.AddEndpointsApiExplorer();
        builder.Services.AddSwaggerGen();

        // Global rate limiting.
        builder.Services.AddRateLimiter(opt =>
        {
            opt.RejectionStatusCode = 429;
            opt.GlobalLimiter =
            PartitionedRateLimiter.Create<HttpContext, string>(
                httpContext => RateLimitPartition.GetFixedWindowLimiter(
                    partitionKey: httpContext.User.Identity?.Name ??
                    httpContext.Request.Headers.Host.ToString(),
                    factory: partition => new
FixedWindowRateLimiterOptions
                    {
                        AutoReplenishment = true,
                        PermitLimit = 4,
                        QueueLimit = 0,
                        Window = TimeSpan.FromSeconds(20)
                    }));
        });

        var app = builder.Build();

        if (app.Environment.IsDevelopment())
        {
            app.UseSwagger();
            app.UseSwaggerUI();
        }
        // Enable rate limiting middleware.
        app.UseRateLimiter();
        app.UseHttpsRedirection();
        app.UseAuthorization();
        app.MapControllers();
        app.Run();
    }
}
```

This means that we don't need to specify that we want to use it on a specific endpoint. I specify that I want to set a global rate limiter, and I can set that global limiter to any of the rate limiters. In Code Listing 44, this is set to a fixed window limiter.

I am also specifying via the options that I only want to allow four requests in the time span of 20 seconds. This fixed window limiter is slightly different from the one we created previously in Code Listing 41. I am specifying here that I want to apply this rate limiter per username with **`httpContext.User.Identity?.Name`**. If not authenticated, then I want to use the hostname with **`httpContext.Request.Headers.Host.ToString()`**.

Because I am creating a global rate limiter, I also do not need to apply it specifically, meaning I can simply use **`app.MapControllers()`**. I have also specified that when a request is rejected, the API needs to return a **`429 Too Many Requests`** status.



```
curl -X 'GET' \
  'https://localhost:7147/WeatherForecast' \
  -H 'accept: text/plain'
```

```
https://localhost:7147/WeatherForecast
```

429
Undocumented
Error: response status is 429

```
content-length: 0
date: Sun,16 Jul 2023 12:17:46 GMT
server: Kestrel
```

Figure 32: Too Many Requests for Global Rate Limiter

Running the API and performing five **`GET`** requests within the 20-second window will return the status **`429`**, as seen in Figure 32.

Chaining rate limiters

Lastly, I want to have a look at chaining rate limiters. In Code Listing 45, I created a chained rate limiter by using **`CreateChained`** and adding two rate limiters.

Code Listing 45: Chaining Rate Limiters

```
public class Program
{
    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);
        builder.Services.AddControllers();
        builder.Services.AddEndpointsApiExplorer();
        builder.Services.AddSwaggerGen();

        // Chaining rate limiters.
        builder.Services.AddRateLimiter(opt =>
        {
            opt.GlobalLimiter =
PartitionedRateLimiter.CreateChained<HttpContext>(

                PartitionedRateLimiter.Create<HttpContext, string>(
                httpContext => RateLimitPartition.GetFixedWindowLimiter(
                    partitionKey: httpContext.User.Identity?.Name ??
                    httpContext.Request.Headers.Host.ToString(),
                    factory: partition => new
FixedWindowRateLimiterOptions
                    {
                        AutoReplenishment = true,
                        PermitLimit = 100,
                        QueueLimit = 0,
                        Window = TimeSpan.FromMinutes(1)
                    })),

                PartitionedRateLimiter.Create<HttpContext, string>(
                httpContext => RateLimitPartition.GetFixedWindowLimiter(
                    partitionKey: httpContext.User.Identity?.Name ??
                    httpContext.Request.Headers.Host.ToString(),
                    factory: partition => new
FixedWindowRateLimiterOptions
                    {
                        AutoReplenishment = true,
                        PermitLimit = 1000,
                        QueueLimit = 0,
                        Window = TimeSpan.FromHours(1)
                    })));
        });

        var app = builder.Build();

        if (app.Environment.IsDevelopment())
        {
            app.UseSwagger();
            app.UseSwaggerUI();
        }

        // Enable rate limiting middleware.
        app.UseRateLimiter();
        app.UseHttpsRedirection();
        app.UseAuthorization();
        app.MapControllers();
        app.Run();
    }
}
```

The first rate limiter tells ASP.NET Core that I only want to allow 100 requests per minute. I do this by specifying the **`PermitLimit = 100`** and the **`Window = TimeSpan.FromMinutes(1)`**.

The second rate limiter specifies that I only want to allow 1,000 requests in a single hour. I do this by specifying **`PermitLimit = 1000`** and **`Window = TimeSpan.FromHours(1)`**.

Therefore, my API can handle bursts of requests but still limit the total number of requests to whatever the second rate limiter allows. My API will then apply the combination of the two rate limiters that I have specified.

Output caching

ASP.NET Core provides developers with various options when it comes to caching. These caching options have also been around for quite some time. These are:

- In-memory cache
- Distributed cache
- Response cache
- Output caching (new in .NET Core 7)

In-memory cache and distributed cache are server-side options. The client does not know about these. Let's have a look at these in more detail.

In-memory cache

The simplest of the caching mechanisms is in-memory cache. Here we simply store the data in the memory of the server. In-memory cache is key-value based. Based on the key, we can return a value stored previously from the cache. Due to the in-memory nature of this cache, this caching mechanism might not work as you expect when using multiple servers, unless you set up server affinity. In situations like this, using a distributed cache, such as Redis, is recommended.

Distributed cache

Multiple servers can access a shared, distributed cache. It, therefore, does not matter if the same server handles the request. Each server has access to the same shared data.

Response cache

When we use response caching, it relies on enabling the client to cache server responses based on specific HTTP cache headers. Sometimes this approach does not work with UI applications such as MVC-based pages because browsers typically set request headers that can prevent caching.

Output caching

Output caching is now available with ASP.NET Core 7. Output caching solves the problem with response caching because it caches the HTTP response on the server. It, therefore, does not rely on the headers of the request. As mentioned earlier, response caching is partly controlled by the client. With output caching, the server is in control of deciding if a cached response is sent back. In ASP.NET Core 7, you can enable output caching through middleware.

This means that the first request received is handled as usual. When a new request is received, the server can decide if a cached response is returned. This reduces the load on the server. Various options are available with output caching to control how the cached responses are stored. Different values can be stored based on a parameter value, query value, header value, and so on.

Let's look at some code to demonstrate how to implement output caching. It’s a bit difficult to illustrate output caching in a book, so you need to run the code to see output caching in action.

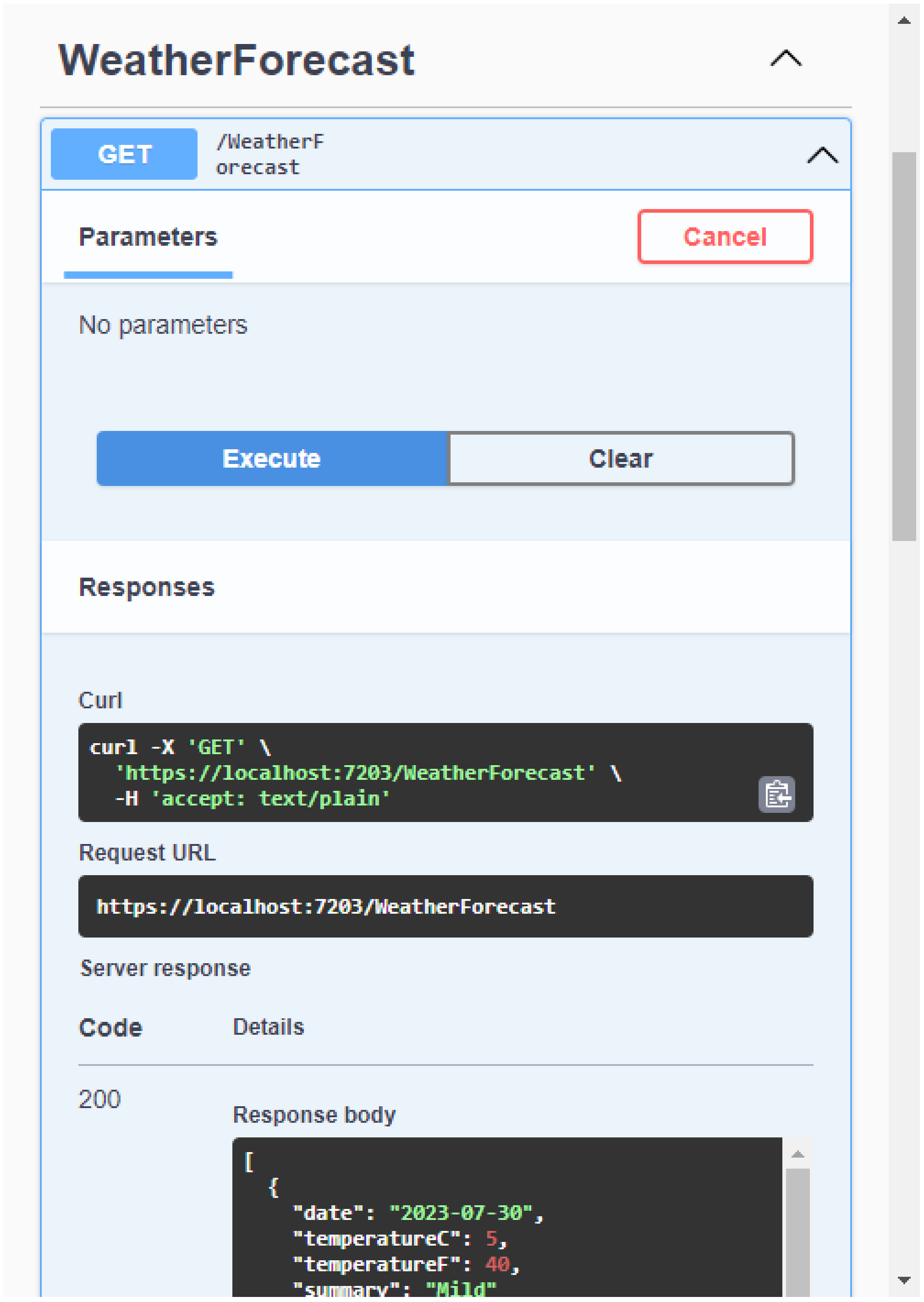
Code Listing 46: Add Output Caching

```
public class Program
{
    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);

        // Add services to the container.
        builder.Services.AddControllers();
        builder.Services.AddEndpointsApiExplorer();
        builder.Services.AddSwaggerGen();

        // Add output cache in the services collection.
        builder.Services.AddOutputCache(opt =>
        {
            opt.DefaultExpirationTimeSpan = TimeSpan.FromSeconds(5);
        });

        var app = builder.Build();

        // Configure the HTTP request pipeline.
        if (app.Environment.IsDevelopment())
        {
            app.UseSwagger();
            app.UseSwaggerUI();
        }

        app.UseHttpsRedirection();

        app.UseAuthorization();
        // Add middleware for output caching.
        app.UseOutputCache();
        // Add CacheOutput for output caching to work.
        app.MapControllers().CacheOutput();

        app.Run();
    }
}
```

The code in Code Listing 46 adds output caching to the services collection with a **DefaultExpirationTimeSpan** of five seconds. This means the API returns the cached response when a request comes in within this five-second time.

Secondly, I must add the middleware **app.UseOutputCache()** for the output cache to work. Output caching is now configured but will not work until you add **CacheOutput()** on the endpoint **MapControllers()**.

Running the API, you will see the Swagger page (Figure 33) where you can test the API.



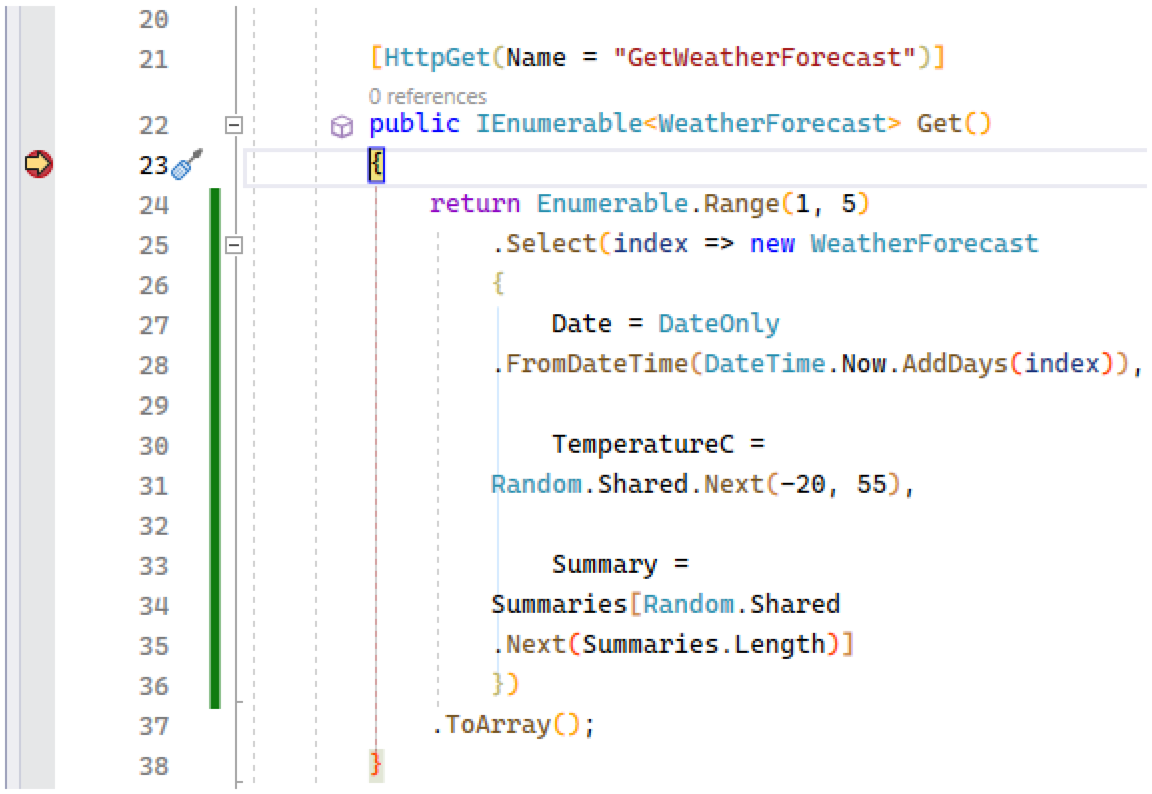
```
curl -X 'GET' \
  'https://localhost:7203/WeatherForecast' \
  -H 'accept: text/plain'
```

```
https://localhost:7203/WeatherForecast
```

```
[
  {
    "date": "2023-07-30",
    "temperatureC": 5,
    "temperatureF": 40,
    "summary": "Mild"
```

Figure 33: Testing Output Caching in Swagger

Clicking on the Execute button will hit the `WeatherForecast` controller on the first request. As seen in Figure 34, you will see the controller hit on the first request if you place a breakpoint as illustrated.



```
[HttpGet(Name = "GetWeatherForecast")]
0 references
public IEnumerable<WeatherForecast> Get()
{
    return Enumerable.Range(1, 5)
        .Select(index => new WeatherForecast
        {
            Date = DateOnly
        .FromDateTime(DateTime.Now.AddDays(index)),

            TemperatureC =
        Random.Shared.Next(-20, 55),

            Summary =
        Summaries[Random.Shared
        .Next(Summaries.Length)]
        })
    .ToArray();
}
```

Figure 34: The WeatherForecast Controller Breakpoint

Subsequent requests within five seconds won't hit the controller. Once the five seconds are over, the controller is hit again.

This approach is a broad one because of how we configured output caching. This results in all controllers caching their responses. We can be more granular when configuring how output caching works.

Imagine that we want to cache the output based on the forecast days, which will be specified in the query string. In Code Listing 47, notice that I have added a policy to vary the cached responses by the query string **days**.

Code Listing 47: Configure Caching by Query Strings

```
public class Program
{
    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);

        // Add services to the container.
        builder.Services.AddControllers();
        builder.Services.AddEndpointsApiExplorer();
        builder.Services.AddSwaggerGen();
        // Add output cache in the services collection.
        builder.Services.AddOutputCache(opt =>
        {
            opt.DefaultExpirationTimeSpan = TimeSpan.FromSeconds(5);
        });

        var app = builder.Build();

        // Configure the HTTP request pipeline.
        if (app.Environment.IsDevelopment())
        {
            app.UseSwagger();
            app.UseSwaggerUI();
        }

        app.UseHttpsRedirection();

        app.UseAuthorization();
        // Add middleware for output caching.
        app.UseOutputCache();
        // Add CacheOutput for output caching to work.
        app.MapControllers().CacheOutput(policy =>
        {
            policy.SetVaryByQuery("days");
            policy.Expire(TimeSpan.FromSeconds(5));
        });

        app.Run();
    }
}
```

I also need to modify the controller action, as seen in Code Listing 48, to use the days I pass it to return the forecast for the days specified.

Code Listing 48: Specify Forecast Days

```
[HttpGet(Name = "GetWeatherForecast")]
public IEnumerable<WeatherForecast> Get(int days)
{
    return Enumerable.Range(1, days).Select(index => new WeatherForecast
        {
            Date = DateOnly.FromDateTime(DateTime.Now.AddDays(index)),

            TemperatureC = Random.Shared.Next(-20, 55),

            Summary = Summaries[Random.Shared.Next(Summaries.Length)]
        })
    .ToArray();
}
```

This means that if I run the API and call it using the URL **`https://localhost:7203/weatherforecast?days=2`**, I receive the following response, as seen in Code Listing 49.

Code Listing 49: Returning a Two-Day Forecast

```
[
  {
    "date": "2023-07-30",
    "temperatureC": -6,
    "temperatureF": 22,
    "summary": "Mild"
  },
  {
    "date": "2023-07-31",
    "temperatureC": 43,
    "temperatureF": 109,
    "summary": "Balmy"
  }
]
```

If I call a six-day forecast using the URL **`https://localhost:7203/weatherforecast?days=6`**, I receive the response illustrated in Code Listing 50. Note that the summary values don't make sense when taking the temperatures into consideration, but this is not important in this example.

Code Listing 50: Returning a Six-Day Forecast

```
[
  {
    "date": "2023-07-30",
    "temperatureC": 2,
    "temperatureF": 35,
    "summary": "Warm"
  },
  {
    "date": "2023-07-31",
    "temperatureC": -18,
    "temperatureF": 0,
    "summary": "Hot"
  },
  {
    "date": "2023-08-01",
    "temperatureC": 25,
    "temperatureF": 76,
    "summary": "Cool"
  },
  {
    "date": "2023-08-02",
    "temperatureC": 42,
    "temperatureF": 107,
    "summary": "Cool"
  },
  {
    "date": "2023-08-03",
    "temperatureC": -7,
    "temperatureF": 20,
    "summary": "Bracing"
  },
  {
    "date": "2023-08-04",
    "temperatureC": 32,
    "temperatureF": 89,
    "summary": "Mild"
  }
]
```

The difference is that both requests are cached based on the query string **`days`**. Subsequent requests within five seconds using the two-day and six-day forecasts return the cached response.

What are minimal APIs?

With .NET 7, there have been some changes to minimal APIs. Introduced in .NET 6, minimal APIs allow developers to create APIs without the notion of controllers. Think of minimal APIs as a model for creating smaller APIs using only a few lines of code.

Minimal APIs rely on top-level statements, and as developers started becoming used to this minimal approach to creating APIs, a few things were still missing. Microsoft, therefore, started maturing the platform to what we now have in .NET 7.

The three main new features are:

- Endpoint filters
- Route groups
- Typed results

Endpoint filters are a way to execute code before and after the logic of the endpoint handler. The filter logic can inspect and change parameters, and using endpoint filters is helpful for things such as logging or validation.

Logically, if you have multiple endpoints, you will need to use endpoint filters on each one. This might result in repeated code, which is why route groups have been introduced. Route groups are used to group endpoints with a common prefix, resulting in less duplication of code.

Let's have a closer look at using minimal APIs with endpoint filters and route groups.

Endpoint filters

In Code Listing 51, I have created a very minimal (see what I did there?) API that returns a collection of movies from a repository I created.

Code Listing 51: The Movie Repository Minimal API

```
using MinimalAPIs.Model;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddAuthorization();
builder.Services.AddScoped<IMovieRepository, MovieRepository>();

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();
app.UseAuthorization();

app.MapGet("/movies", (IMovieRepository movieRepo) =>
{
    return movieRepo.GetAllMovieItems();
});

app.Run();
```

Running the minimal API and calling the **`movies`** endpoint returns the movies in my repository, as seen in Code Listing 52.

Code Listing 52: The Returned Movie Collection

```
[
    {
      "title": "Movie 1",
      "description": "Movie 1 description.",
        "watched": true
    },
    {
        "title": "Movie 2",
        "description": "Movie 2 description.",
        "watched": false
    },
    {
        "title": "Movie 3",
        "description": "Movie 3 description.",
        "watched": true
    },
    {
        "title": "Movie 4",
        "description": "Movie 4 description.",
        "watched": false
    }
]
```

Let’s extend this with an endpoint filter by adding the **`AddEndpointFilter`** method, as seen in Code Listing 53. The **`EndpointFilterInvocationContext context`** gives us access to the HTTP context, which then allows us to access the request.

Code Listing 53: Adding an Endpoint Filter

```
app.MapGet("/movies", (IMovieRepository movieRepo) =>
{
    return movieRepo.GetAllMovieItems();
}).AddEndpointFilter(async (context, next) =>
{
    app.Logger.LogInformation($"Request
{context.HttpContext.Request.Path} processed at {DateTime.Now}");

    var result = await next(context);

    app.Logger.LogInformation($"Request
{context.HttpContext.Request.Path} processed at {DateTime.Now}");

    return result;
});
```

I have added logging by including the lines of code seen in Code Listing 54.

Code Listing 54: Adding Logging

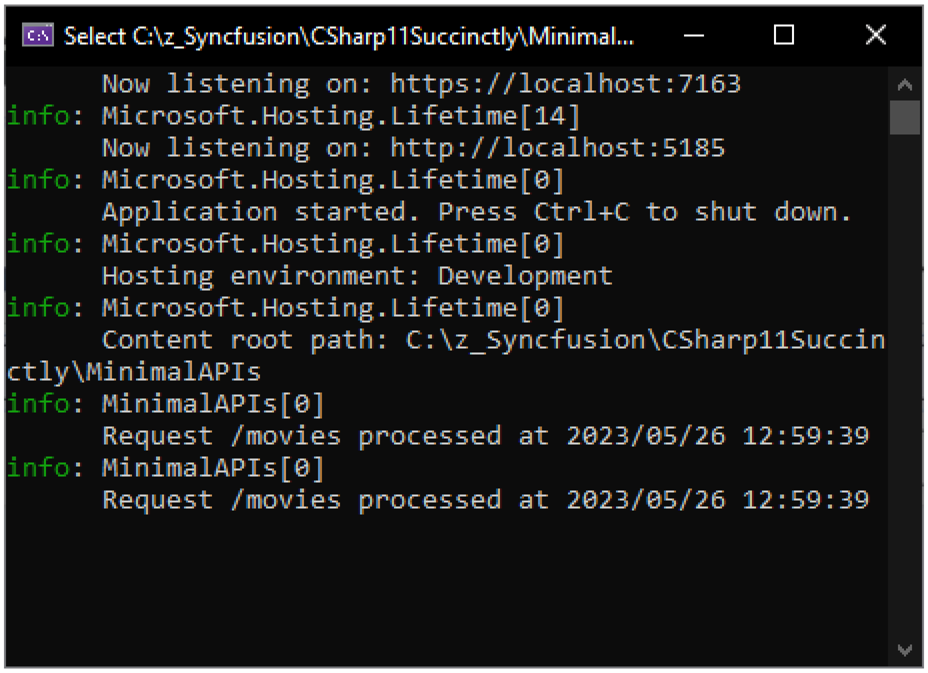
```
builder.Logging.ClearProviders();
builder.Logging.AddConsole();
builder.Logging.AddDebug();
```

The complete code is illustrated in Code Listing 55.

Code Listing 55: The Complete Code

```
using MinimalAPIs.Model;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddAuthorization();
builder.Services.AddScoped<IMovieRepository, MovieRepository>();

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

builder.Logging.ClearProviders();
builder.Logging.AddConsole();
builder.Logging.AddDebug();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();
app.UseAuthorization();

app.MapGet("/movies", (IMovieRepository movieRepo) =>
{
    return movieRepo.GetAllMovieItems();
}).AddEndpointFilter(async (context, next) =>
{
    app.Logger.LogInformation($"Request
{context.HttpContext.Request.Path} processed at {DateTime.Now}");

    var result = await next(context);

    app.Logger.LogInformation($"Request
{context.HttpContext.Request.Path} processed at {DateTime.Now}");

    return result;
});

app.Run();
```

The endpoint filter will now add a log entry before execution. Then, **`await next`** executes the next filter (if any), and I write to the log again after that.



Select C:\z_Syncfusion\CSharp11Succinctly\Minimal...

```
      Now listening on: https://localhost:7163
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5185
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\z_Syncfusion\CSharp11Succin
ctly\MinimalAPIs
info: MinimalAPIs[0]
      Request /movies processed at 2023/05/26 12:59:39
info: MinimalAPIs[0]
      Request /movies processed at 2023/05/26 12:59:39
```

Figure 35: The Log Output

When I run the API and execute the request, the log entries are seen in the console window, as seen in Figure 35. The endpoint filter is therefore working correctly, adding log entries to the configured logging providers.

In Code Listing 56, I have added a **`POST`** to create movies in the movie repository. This **`MapPost`** endpoint also implements an endpoint filter.

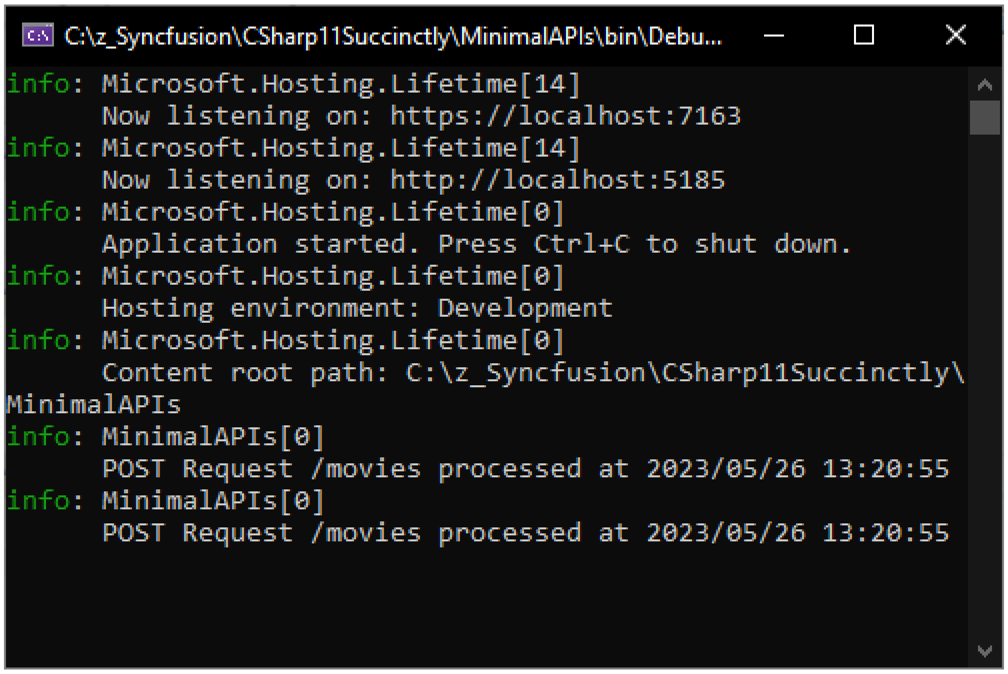
Code Listing 56: Adding a POST Request

```
using MinimalAPIs.Model;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddAuthorization();
builder.Services.AddScoped<IMovieRepository, MovieRepository>();

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

builder.Logging.ClearProviders();
builder.Logging.AddConsole();
builder.Logging.AddDebug();
```

```
var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();
app.UseAuthorization();

app.MapGet("/movies", (IMovieRepository movieRepo) =>
{
    return movieRepo.GetAllMovieItems();
}).AddEndpointFilter(async (context, next) =>
{
    app.Logger.LogInformation($"Request
{context.HttpContext.Request.Path} processed at {DateTime.Now}");

    var result = await next(context);

    app.Logger.LogInformation($"Request
{context.HttpContext.Request.Path} processed at {DateTime.Now}");

    return result;
});

app.MapPost("/movies", (MovieItem movieItem, IMovieRepository movieRepo)
=>
{
    if (movieItem is null) return Results.BadRequest();
    if (string.IsNullOrEmpty(movieItem.Title)) return
Results.BadRequest();

    movieRepo.AddMovieItem(movieItem);

    return Results.Ok();

}).AddEndpointFilter(async (context, next) =>
{
    app.Logger.LogInformation($"POST Request
{context.HttpContext.Request.Path} processed at {DateTime.Now}");

    var result = await next(context);

    app.Logger.LogInformation($"POST Request
{context.HttpContext.Request.Path} processed at {DateTime.Now}");
```

```
    return result;
});

app.Run();
```

Rerunning the API and adding a movie will, as expected, log the information to the console.



```
C:\z_Syncfusion\CSharp11Succinctly\MinimalAPIs\bin\Debu...
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:7163
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5185
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\z_Syncfusion\CSharp11Succinctly\
MinimalAPIs
info: MinimalAPIs[0]
      POST Request /movies processed at 2023/05/26 13:20:55
info: MinimalAPIs[0]
      POST Request /movies processed at 2023/05/26 13:20:55
```

Figure 36: The Log Output for the POST

The problem, however, is that now we have repeated code because we have two endpoint filters. We have one on the **`GET`** request, and another on the **`POST`** request.

Let's improve the code by adding the generic **`AddEndpointFilter`** implementation. This is illustrated in Code Listing 57, where I have added **`AddEndpointFilter<MinimalAPILogFilter>()`** to my **`MapGet`** and **`MapPost`**. This tells my API that I want to use my **`MinimalAPILogFilter`** as an endpoint filter.

Code Listing 57: Adding the Generic AddEndpointFilter Implementation

```
using MinimalAPIs;
using MinimalAPIs.Model;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddAuthorization();
builder.Services.AddScoped<IMovieRepository, MovieRepository>();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

builder.Logging.ClearProviders();
builder.Logging.AddConsole();
builder.Logging.AddDebug();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();
app.UseAuthorization();

app.MapGet("/movies", (IMovieRepository movieRepo) =>
{
    return movieRepo.GetAllMovieItems();
}).AddEndpointFilter<MinimalAPILogFilter>();

app.MapPost("/movies", (MovieItem movieItem, IMovieRepository movieRepo)
=>
{
    if (movieItem is null) return Results.BadRequest();
    if (string.IsNullOrEmpty(movieItem.Title)) return
Results.BadRequest();

    movieRepo.AddMovieItem(movieItem);

    return Results.Ok();

}).AddEndpointFilter<MinimalAPILogFilter>();


app.Run();
```

The **`MinimalAPILogFilter`** is a custom class that contains my logging code inside the delegate. You can see the code for my custom class in Code Listing 58. It implements the **`IEndpointFilter`** interface and contains the logging logic in the **`InvokeAsync`** method.

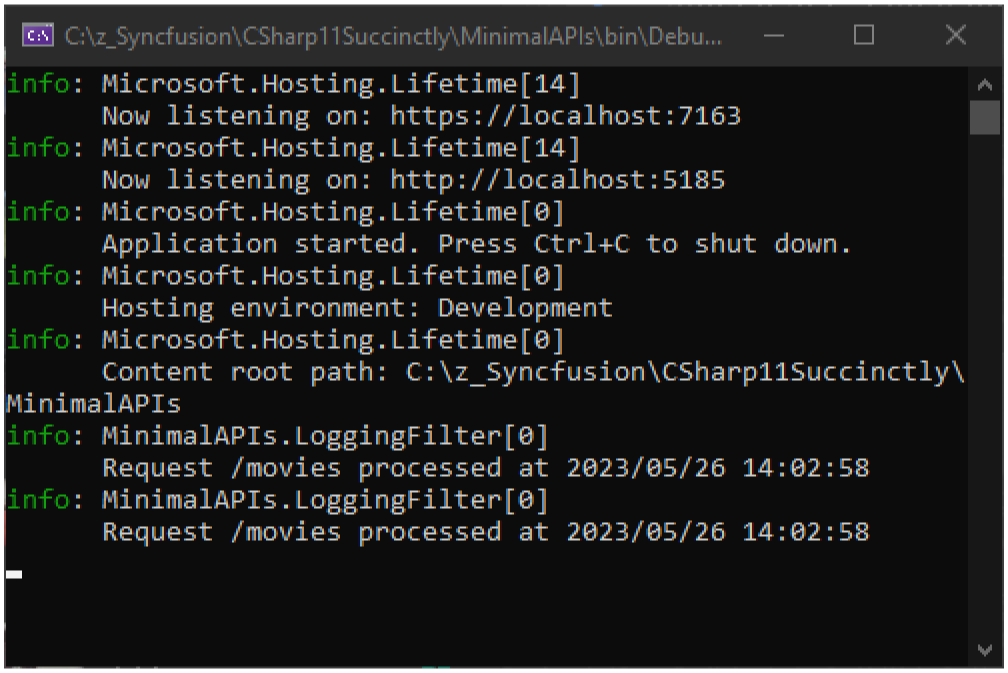
Dependency injection provides access to the logger, which I inject via the **`LoggingFilter`** constructor.

Code Listing 58: The Custom MinimalAPILogFilter Class

```
namespace MinimalAPIs;
public class LoggingFilter : IEndpointFilter
{
    protected readonly ILogger _logger;
    protected LoggingFilter(ILoggerFactory loggerFac)
    {
        _logger = loggerFac.CreateLogger<LoggingFilter>();
    }
    public virtual async ValueTask<object?>
InvokeAsync(EndpointFilterInvocationContext context,
EndpointFilterDelegate next)
    {
        _logger.LogInformation($"Request
{context.HttpContext.Request.Path} processed at {DateTime.Now}");

        var result = await next(context);

        _logger.LogInformation($"Request
{context.HttpContext.Request.Path} processed at {DateTime.Now}");

        return result;
    }
}

class MinimalAPILogFilter : LoggingFilter
{
    public MinimalAPILogFilter(ILoggerFactory loggerFac) :
base(loggerFac) { }
}
```

Lastly, I created a derived class called **`MinimalAPILogFilter`** that I use in the generic **`AddEndpointFilter`** implementation.

Running the API and performing a **`GET`** or a **`POST`** request still logs to the console as expected, as seen in Figure 37.



```
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:7163
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5185
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\z_Syncfusion\CSharp11Succinctly\
MinimalAPIs
info: MinimalAPIs.LoggingFilter[0]
      Request /movies processed at 2023/05/26 14:02:58
info: MinimalAPIs.LoggingFilter[0]
      Request /movies processed at 2023/05/26 14:02:58
```

Figure 37: The Console Log Output Still Works

Creating a custom class is quite convenient and reduces the repeated code we need to write to implement endpoint filters. However, having to configure each endpoint to use the generic **`AddEndpointFilter`** implementation might become a bit of a pain.

It is now possible to use route groups. Let's see how that works next.

Route groups

Using a common prefix, we can now group routes. On this group level, we can add code that is then applied to all the endpoints contained in that group.

Consider the code illustrated in Code Listing 59.

Code Listing 59: Adding Route Groups

```
using MinimalAPIs.Model;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddAuthorization();
builder.Services.AddScoped<IMovieRepository, MovieRepository>();

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
builder.Logging.ClearProviders();
builder.Logging.AddConsole();
builder.Logging.AddDebug();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();
app.UseAuthorization();

var groupedLogging = app.MapGroup("/api").AddEndpointFilter(async
(context, next) =>
{
app.Logger.LogInformation($"Request {context.HttpContext.Request.Path}
processed at {DateTime.Now}");

var result = await next(context);

app.Logger.LogInformation($"Request {context.HttpContext.Request.Path}
processed at {DateTime.Now}");

return result;
});


groupedLogging.MapGet("/movies", (IMovieRepository movieRepo) =>
{
    return movieRepo.GetAllMovieItems();
});

groupedLogging.MapPost("/movies", (MovieItem movieItem, IMovieRepository
movieRepo) =>
{
    if (movieItem is null) return Results.BadRequest();
    if (string.IsNullOrEmpty(movieItem.Title)) return
Results.BadRequest();

    movieRepo.AddMovieItem(movieItem);

    return Results.Ok();

});
```

```
app.Run();
```

I created a variable called **`groupedLogging`**, which is of type **`RouteGroupBuilder`**, and assigned a **`MapGroup`** that specifies **`/api`**. This means that all endpoints starting with **`/api`** will get this configuration.

Lastly, I have to modify my endpoints using **`app.MapGet`** and **`app.MapPost`** to **`groupedLogging.MapGet`** and **`groupedLogging.MapPost`**.

This means I no longer need to apply the endpoint filter on each endpoint individually.

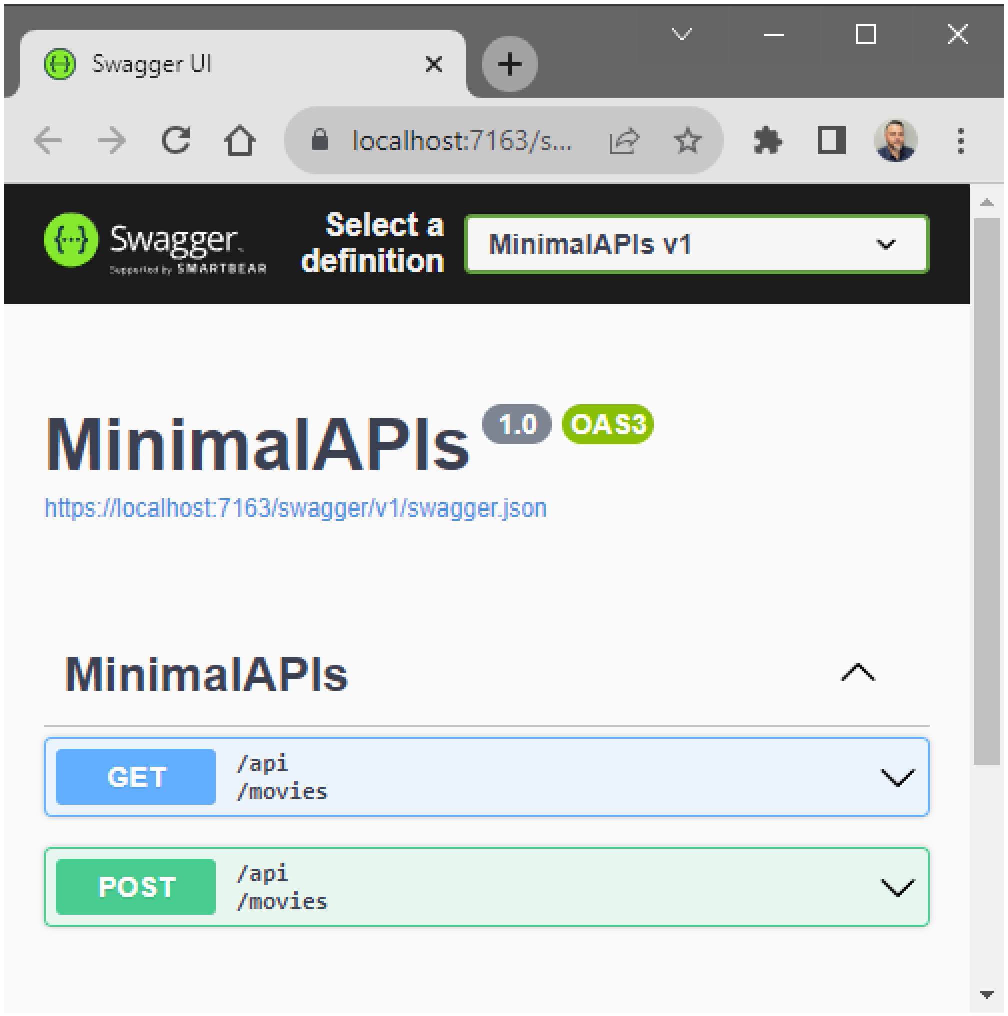
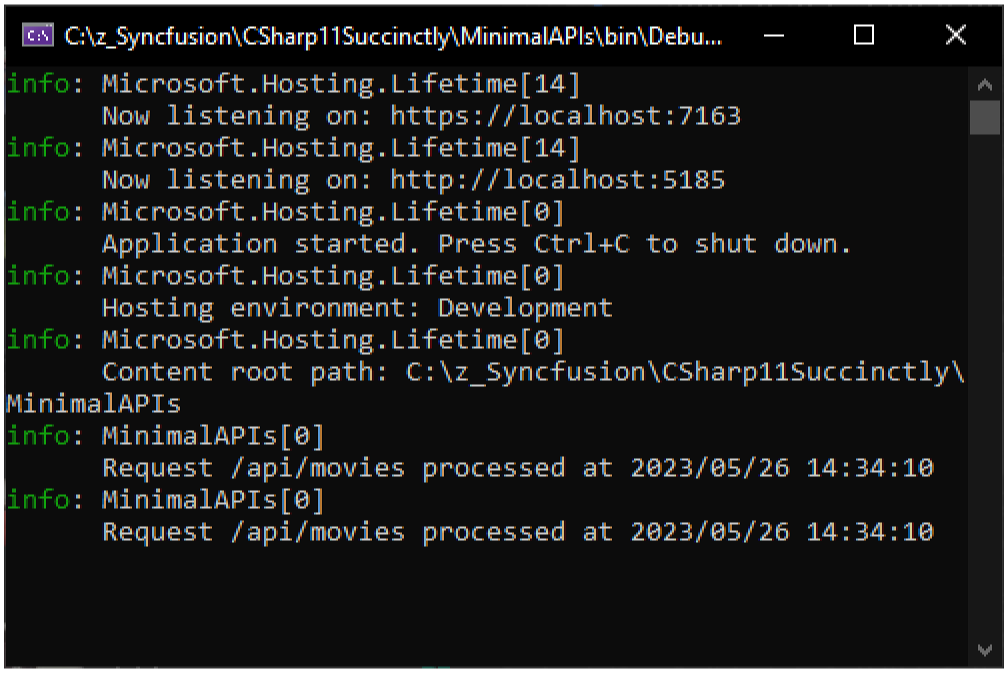


Figure 38: Swagger now Shows /api/movies

Running my API and looking at the endpoints in Swagger, you will notice that my endpoint now says **`/api/movies`**, as seen in Figure 38.



```
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:7163
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5185
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\z_Syncfusion\CSharp11Succinctly\
MinimalAPIs
info: MinimalAPIs[0]
      Request /api/movies processed at 2023/05/26 14:34:10
info: MinimalAPIs[0]
      Request /api/movies processed at 2023/05/26 14:34:10
```

Figure 39: The Console Log Displaying the Log Information

Executing either a **GET** or **POST** request still logs the information to my console as before. This means that the endpoint filter is still working as expected.

Using static abstract interface members

I found some interesting examples on the internet of using static abstract interfaces. Earlier in the book, we had a look at Generic Math support and how static abstract interface members enable this feature. I mentioned that we would look at how you can apply this in a real-world situation. Minimal APIs provide an opportunity to illustrate how static abstract interface members can be used to make life much easier.

Using Minimal APIs does not necessarily mean I need to keep all my logic in the Program.cs file. In fact, I want to keep my Program.cs file as lean as possible. For this reason, I am restructuring my Minimal API into something more manageable. To accomplish this, I am creating classes that simplify my endpoints.

Code Listing 60: The IEndpoint Interface

```
namespace MinimalAPIsStaticAbstrInterfMembers;
public interface IEndpoint
{
    static abstract string Pattern { get; }
    static abstract HttpMethod Method { get; }
    static abstract Delegate Handler { get; }
}
```

You will see an **`Interface`** in Code Listing 60 that uses a few static, abstract **`Interface`** members. It contains properties for a **`Pattern`** (which is the path to the endpoint); a **`Method`** property that specifies **`GET`**, **`POST`**, **`PUT`**, and so on; and lastly, there is a **`Delegate`**, which is the handler that tells my API how to handle the request it receives.

Code Listing 61: MapEndPoint Extension Method

```
namespace MinimalAPIsStaticAbstrInterfMembers;
public static class EndpointExtensions
{
    public static IEndpointRouteBuilder MapEndPoint<T>(this
IEndpointRouteBuilder app) where T : IEndpoint
    {
        app.MapMethods(
            T.Pattern
            , new[] { T.Method.ToString() }
            , T.Handler
            );
        return app;
    }
}
```

In Code Listing 61, you see an extension method that maps my endpoint. This extension method has a constraint on **`T`** to ensure that it implements **`IEndpoint`**. It then maps the **`Pattern`**, **`Method`**, and **`Handler`** and returns an **`IEndpointRouteBuilder`**.

Code Listing 62: Get Movies Endpoint Class

```
using MinimalAPIs.Model;

namespace MinimalAPIsStaticAbstrInterfMembers;
public class GetMoviesEndpoint : IEndpoint
{
    public static string Pattern => "movies";

    public static HttpMethod Method => HttpMethod.Get;

    public static Delegate Handler => GetMovies;

    private static IResult GetMovies(IMovieRepository repo)
    {
        return Results.Ok(repo.GetAllMovieItems());
    }
}
```

I can now create a **`GetMoviesEndpoint`** class (Code Listing 62) that implements the **`IEndpoint`** interface and returns the movies in the movie repository.

The Program.cs file in Code Listing 63 is much cleaner because I do not have to keep the minimal API logic in here. I am now able to call my endpoints by using **`app.MapEndPoint`** and provide it with the endpoint class created earlier.

I am now in a position to structure my minimal API more efficiently, which allows me to better manage my endpoint logic.

Code Listing 63: A Cleaner Program.cs Class

```
using MinimalAPIs.Model;

namespace MinimalAPIsStaticAbstrInterfMembers;

public class Program
{
    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);

        // Add services to the container.
        builder.Services.AddAuthorization();
        builder.Services.AddScoped<IMovieRepository, MovieRepository>();

        builder.Services.AddEndpointsApiExplorer();
        builder.Services.AddSwaggerGen();

        var app = builder.Build();

        // Configure the HTTP request pipeline.
        if (app.Environment.IsDevelopment())
        {
            app.UseSwagger();
            app.UseSwaggerUI();
        }

        app.UseHttpsRedirection();
        app.UseAuthorization();

        app.MapEndPoint<GetMoviesEndpoint>();

        app.Run();
    }
}
```

This was made possible by using static abstract interface members, keeping my code more succinct and structured.

In conclusion

This chapter focused on ASP.NET Core 7. The features introduced make developers' lives much easier. .NET 7, however, has so much more to offer. Without writing a single line of code, you can optimize your codebase by upgrading to .NET 7.

In the next section, we will briefly look at LINQ and JSON serialization improvements. There are many more bundled with .NET 7, but it would take a book on its own to outline all the performance improvements available.

Chapter 4 Performance Improvements

The performance well has by no means run dry. Even though .NET 6 was fast, and one could imagine that there was little else to do to speed up .NET any more, along comes .NET 7, with some serious performance improvements. To quantify just how many performance improvements there are, we need to refer back to an article Stephen Toub (a developer on the .NET team at Microsoft) wrote in August of 2022.

Each time he reviewed a PR that might positively impact performance, he copied that link to a journal. When he reviewed the journal for the article, he discovered that almost a thousand performance-impacting pull requests went into the release of .NET 7.

We can safely assume that .NET 7 is very fast. Let's look at just how fast it is in the following sections.

Benchmarking .NET 6 vs. .NET 7

Illustrating the performance improvements in .NET 7, we need to look at setting up a simple Console application that includes the NuGet package BenchmarkDotNet, as seen in Figure 40.

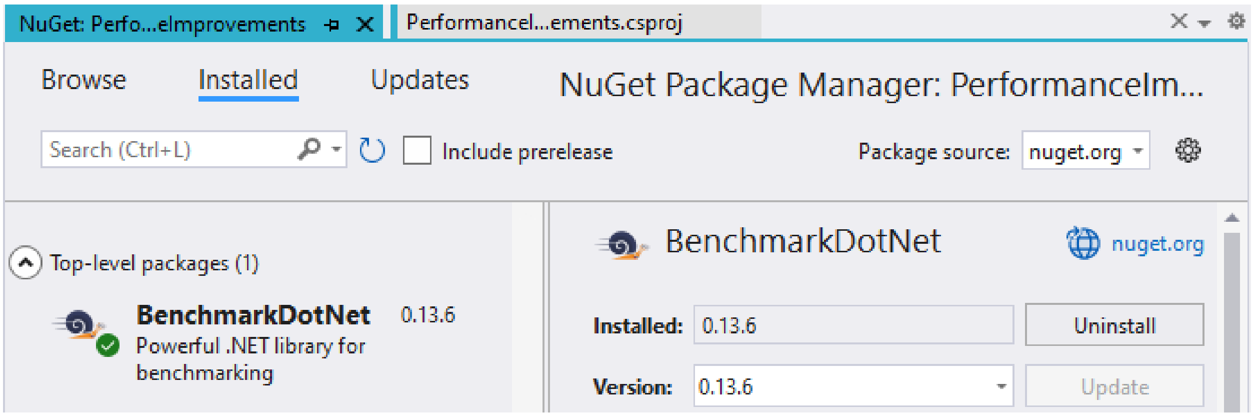


Figure 40: BenchmarkDotNet

To compare the improvements in .NET 7, we benchmark it against .NET 6 to illustrate how the same code performs across both .NET versions. We must therefore target multiple frameworks in our Console application.

The easiest way to do this is to modify the csproj file of the Console application used for benchmarking.

You can see how my csproj file looks in Code Listing 64.

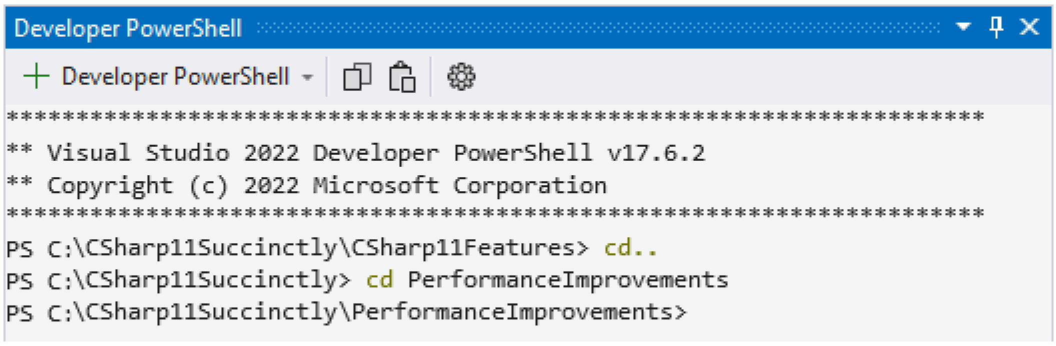
Code Listing 64: The csproj File Targeting .NET 6 and .NET 7

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFrameworks>net7.0;net6.0</TargetFrameworks>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="benchmarkdotnet" Version="0.13.6" />
  </ItemGroup>

</Project>
```

Next, you need to open up the Terminal window in Visual Studio. You can do this by clicking **View** > **Terminal** in the Visual Studio menu, or you can hold down **`Ctrl + ``** to open up the Developer PowerShell.

Note: The menu says** Terminal**, but the opened window title says** Developer PowerShell**.

We will run the benchmarking project from the command line, so having the Terminal window open will make it easier.



```
**********************************************************************
** Visual Studio 2022 Developer PowerShell v17.6.2
** Copyright (c) 2022 Microsoft Corporation
**********************************************************************
PS C:\CSharp11Succinctly\CSharp11Features> cd..
PS C:\CSharp11Succinctly> cd PerformanceImprovements
PS C:\CSharp11Succinctly\PerformanceImprovements>
```

Figure 41: Developer PowerShell in Visual Studio

With this window open, change the directory to the benchmarking project. In my example, you can find the code in a project called **`PerformanceImprovements`** in the main solution. We are ready to write some code. Let's get started.

LINQ—improvements to existing operations

By now, most developers are familiar with LINQ. It allows us to express complicated operations concisely using the LINQ syntax. This expressivity does come with a bit of an overhead cost. This overhead cost leads some developers to avoid using LINQ in their code, but LINQ has its place and is extremely useful as long as you are cognizant of where you use it in your code.

With .NET 7, these improvements to performance come to LINQ out of the box. It means that merely upgrading your solution from .NET 6 to .NET 7 gives it a performance boost without you having to write a single line of optimization code.

Code Listing 65: LINQ Example on Existing Operations

```
using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Running;

namespace PerformanceImprovements;

public class Program
{
    static void Main(string[] args) => BenchmarkRunner.Run<Bench>();
}

[MemoryDiagnoser(displayGenColumns: false)]
[DisassemblyDiagnoser]
[HideColumns("Error", "StdDev", "Median", "RatioSD")]
public class Bench
{

    private IEnumerable<float> _randFloats = CreateRandomFloats();

    [Benchmark]
    public float Min() => _randFloats.Min();

    [Benchmark]
    public float Max() => _randFloats.Max();

    [Benchmark]
    public float Average() => _randFloats.Average();

    [Benchmark]
    public float Sum() => _randFloats.Sum();

    private static float[] CreateRandomFloats()
    {
        var r = new Random(45);
        var results = new float[100_000];
        for (int i = 0; i < results.Length; i++)
        {
            results[i] = (float)r.NextDouble();
        }
        return results;
    }
}
```

Consider the code in Code Listing 65 that creates a collection of 100,000 random floating-point numbers and then performs calculations on that collection via the LINQ extension methods **`Min`**, **`Max`**, **`Average`**, and **`Sum`**.

In the Terminal window, run the command **`dotnet run -c Release -f net6.0 --filter '**' --runtimes net6.0`** to build the benchmarks in the release configuration targeting .NET 6 and run the project on .NET 6. The benchmarking might take a few minutes, and you see a summary of the result when it completes (Code Listing 66).

Code Listing 66: Benchmark Results for .NET 6

```
[Host]     : .NET 6.0.16 (6.0.1623.17311), X64 RyuJIT AVX2
DefaultJob : .NET 6.0.16 (6.0.1623.17311), X64 RyuJIT AVX2


|  Method |     Mean | Code Size | Allocated |
|-------- |---------:|----------:|----------:|
|     Min | 636.9 us |     347 B |      33 B |
|     Max | 592.4 us |     411 B |      33 B |
| Average | 541.8 us |     343 B |      33 B |
|     Sum | 550.9 us |     258 B |      33 B |
```

Next, run the command **`dotnet run -c Release -f net7.0 --filter '**' --runtimes net7.0`** to build the benchmarks in the release configuration targeting .NET 7 and run the project on .NET 7. When it completes, you see a similar summary (Code Listing 67), only this time, using .NET 7, the results are very different.

Code Listing 67: Benchmark Results for .NET 7

```
[Host]     : .NET 7.0.5 (7.0.523.17405), X64 RyuJIT AVX2
DefaultJob : .NET 7.0.5 (7.0.523.17405), X64 RyuJIT AVX2


|  Method |     Mean | Code Size | Allocated |
|-------- |---------:|----------:|----------:|
|     Min | 101.1 us |     518 B |         - |
|     Max | 122.5 us |     629 B |         - |
| Average | 200.7 us |     627 B |         - |
|     Sum | 191.6 us |     420 B |         - |
```

To make sense of the results, we must first define what we see here. The following is true:

- Mean: Arithmetic mean of all the measurements
- Allocated: Allocated memory per single operation (managed only, inclusive, 1KB = 1024B)
- 1 ns: 1 Nanosecond
- 1 us: 1 Microsecond

Comparing the results from .NET 6 with those of .NET 7, we can see that the mean times reduce significantly when using .NET 7, and using .NET 7 also does not create any allocations.

While the improvements in the previous example are introduced in existing operations, performance improvements can result in using new APIs in place of the previous ones. Let us look at the popular **`OrderBY`** LINQ method next.

LINQ—using the new Order method

The **`OrderBy`** method in LINQ enables the creation of a sorted copy of the input enumerable. Code Listing 68 lists the benchmarking code required for the next example.

Code Listing 68: LINQ Example Using the New Order() Method

```
using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Running;

namespace PerformanceImprovements;

public class Program
{
    static void Main(string[] args) => BenchmarkRunner.Run<Bench>();
}

[MemoryDiagnoser(displayGenColumns: false)]
[DisassemblyDiagnoser]
[HideColumns("Error", "StdDev", "Median", "RatioSD")]
public class Bench
{

    [Params(1024)]
    public int Length { get; set; }

    private int[] _intArray;

    [GlobalSetup]
    public void Setup() => _intArray = Enumerable.Range(1,
Length).Reverse().ToArray();

    [Benchmark(Baseline = true)]
    public void OrderBy()
    {
        foreach (int _ in _intArray.OrderBy(x => x)) { }
    }

    [Benchmark]
    public void Order()
    {
        foreach (int _ in _intArray.Order()) { }
    }
}
```

You can see that I have an array of integers that I sort using **`OrderBy(x => x)`** and another new **`Order()`** method available in .NET 7. Incidentally, .NET 7 also introduces **`OrderDescending()`**.

If you try and run this benchmark code using **`dotnet run -c Release -f net6.0 --filter '**' --runtimes net6.0`**, you receive a build error. You see this error because the **`Order()`** method is unavailable in .NET 6.

With that in mind, run the Console application using .NET 7 by changing the command to **`dotnet run -c Release -f net7.0 --filter '**' --runtimes net7.0`**.

Code Listing 69: OrderBy() Versus Order() Results

```
|  Method | Length |      Mean | Ratio | Code Size | Allocated | Alloc Ratio |
|-------- |------- |---------:|------:|----------:|----------:|------------:|
| OrderBy |   1024 | 95.13 us |  1.00 |     427 B |   12.3 KB |        1.00 |
|   Order |   1024 | 90.64 us |  0.96 |     336 B |   8.28 KB |        0.67 |
```

The results in Code Listing 69 speak for themselves. With **`OrderBy()`**, the caller passes a **`Func<TSource, TKey>`** predicate, which is used to extract a comparison key for each item. Seeing as it is pretty common to sort items with themselves as the keys, .NET 7 performs the same sorting operation with an implicit **`x => x`** done on behalf of the caller.

If you face a situation where you need to use **`OrderBy(x => x)`**, consider using the **`Order()`** method instead. As you can see from Code Listing 69, the allocations are reduced significantly in .NET 7 using the new **`Order()`** method.

JSON improvements

.NET Core 3.0 introduced developers to **`System.Text.Json`** with a significant investment going into .NET 7. One of the pitfalls, though, concerns how the library caches data. To improve serialization and deserialization performance when the source generator isn't used, reflection is used by **`System.Text.Json`** to emit custom code for reading and writing members of the processed types.

The library incurs a more significant one-time cost performing this code generation, making subsequent handling of the types fast (assuming that the generated code is available for use). The generated handlers require them to be stored somewhere, which is where **JsonSerialierOptions** come into play.

The idea behind **JsonSerializerOptions** is that developers would create an instance once and pass that around to all the required serialization and deserialization calls. If, however, developers deviate from the recommended model, performance tanks. What happens now is that instead of taking the hit for the reflection invoke costs, every new instance of **JsonSerializerOptions** results in a regeneration of the handlers via reflection emit, which in turn increases the cost of the serialization and deserialization.

Consider the code illustrated in Code Listing 70.

Code Listing 70: JSON Serialization

```
using System.Text.Json;
using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Running;

namespace PerformanceImprovements;

public class Program
{
    static void Main(string[] args) => BenchmarkRunner.Run<Bench>();
}

[MemoryDiagnoser(displayGenColumns: false)]
[DisassemblyDiagnoser]
[HideColumns("Error", "StdDev", "Median", "RatioSD")]
public class Bench
{
    private JsonSerializerOptions _options = new JsonSerializerOptions();
    private StudentPoco _student = new StudentPoco();

    [Benchmark(Baseline = true)]
    public string ImplicitOptions() =>
JsonSerializer.Serialize(_student);

    [Benchmark]
    public string WithCached() => JsonSerializer.Serialize(_student,
_options);

    [Benchmark]
    public string WithoutCached() => JsonSerializer.Serialize(_student,
new JsonSerializerOptions());

}

public class StudentPoco
```

```
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
}
```

Running the benchmarks using .NET 6 produces the results in Code Listing 71.

Code Listing 71: JSON Serialization with .NET 6

```
[Host]     : .NET 6.0.16 (6.0.1623.17311), X64 RyuJIT AVX2
DefaultJob : .NET 6.0.16 (6.0.1623.17311), X64 RyuJIT AVX2


|          Method |         Mean | Code Size | Allocated | Alloc Ratio |
|---------------- |-------------:|----------:|----------:|------------:|
| ImplicitOptions |     325.3 ns |     999 B |     232 B |        1.00 |
|      WithCached |     313.1 ns |     871 B |     232 B |        1.00 |
|   WithoutCached | 344,578.1 ns |   1,180 B |   13623 B |       58.72 |
```

Running the benchmarks using .NET 7, illustrated in Code Listing 72, paints a very different picture.

Code Listing 72: JSON Serialization with .NET 7

```
[Host]     : .NET 7.0.5 (7.0.523.17405), X64 RyuJIT AVX2
DefaultJob : .NET 7.0.5 (7.0.523.17405), X64 RyuJIT AVX2


|          Method |       Mean | Code Size | Allocated | Alloc Ratio |
|---------------- |-----------:|----------:|----------:|------------:|
| ImplicitOptions |   323.4 ns |   1,004 B |      80 B |        1.00 |
|      WithCached |   327.7 ns |     680 B |      80 B |        1.00 |
|   WithoutCached | 1,112.5 ns |     865 B |     311 B |        3.89 |
```

.NET 7 adds a global cache containing the type information and separates it from the options instances. Using **`JsonSerializerOptions`** still does cache, but if new handlers are generated with reflection emit, those are cached at the global level.

As seen in the results for **`WithoutCached()`**, creating a **`new JsonSerializerOptions`** is still expensive, but much less expensive when using .NET 7.

In conclusion

Illustrating all the performance improvements made in .NET 7 would take a whole book on its own. The takeaway here is to realize that even though .NET 7 is a Standard Term Support release, that doesn't mean you shouldn't upgrade your projects to .NET 7. The performance boost that your projects gain out of the box is reason enough to upgrade to .NET 7.

For more information on BenchmarkDotNet, refer to their documentation here.