.NET 8 AND C#12

SUCCINCTLY®

BY **DIRK STRAUSS**

SUCCINCTLY® EBOOK SERIES

Syncfusion®

.NET 8 and C# 12 Succinctly®

Dirk Strauss

Foreword by Daniel Jebaraj

ISBN: 978-1-64200-243-0

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 8

Microsoft began working on .NET in 1998, and it has since evolved into a *tour de force* among developers worldwide. From its birth in 2002, .NET has advanced to keep up with the changing demands of the IT landscape, much like biological species evolve to adapt to their surroundings. Introduced as the .NET Framework, it offered a rich set of libraries and tools, becoming popular with an army of software developers globally.

In 2016, entering the .NET Core years, Microsoft announced the open-source, cross-platform implementation of the .NET Framework, refactored and renamed .NET Core. For us old enough to remember the first incarnations of the .NET Framework, the idea of .NET Core seemed almost magical. Enabling developers to build and run apps on Windows, macOS, and Linux, .NET Core quickly gained adoption in the industry.

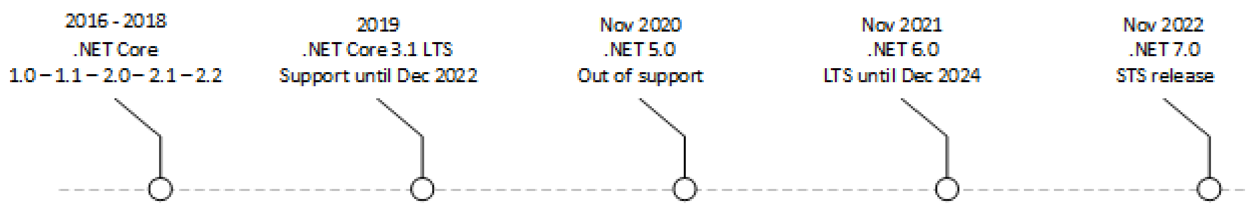


Figure 1: The Jump in Versions

Microsoft finally unified the .NET ecosystem with the release of .NET 5 in 2020. Looking at Figure 1, you will notice that there seems to be a jump between .NET Core 3.1 and .NET 5. 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.

.NET Core 1.0 was released in 2016 and heralded the start of what we know today as .NET 8. Before this release, developers were accustomed to the .NET Framework, released in the early 2000s. It was closed-source and only ran on Windows. .NET Core 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. It was, therefore, a very popular version of .NET Core. Then, in 2020, Microsoft surprised us all with the release of .NET 5.0. Support for .NET 5.0 ended in May 2022.

The History of .NET

2002 - .NET Framework 1.0

The Euro (coins and banknotes) is introduced as a currency in 12 European countries. The .NET Framework 1.0 and C# 1.0 are released.

2005 - .NET Framework 2.0

YouTube is launched in February, transforming the media landscape forever. Later that year, the .NET Framework 2.0 and C# 2.0 are released that included features such as partial classes, nullable types, anonymous methods, iterators, generics, private setters in properties, and static classes.

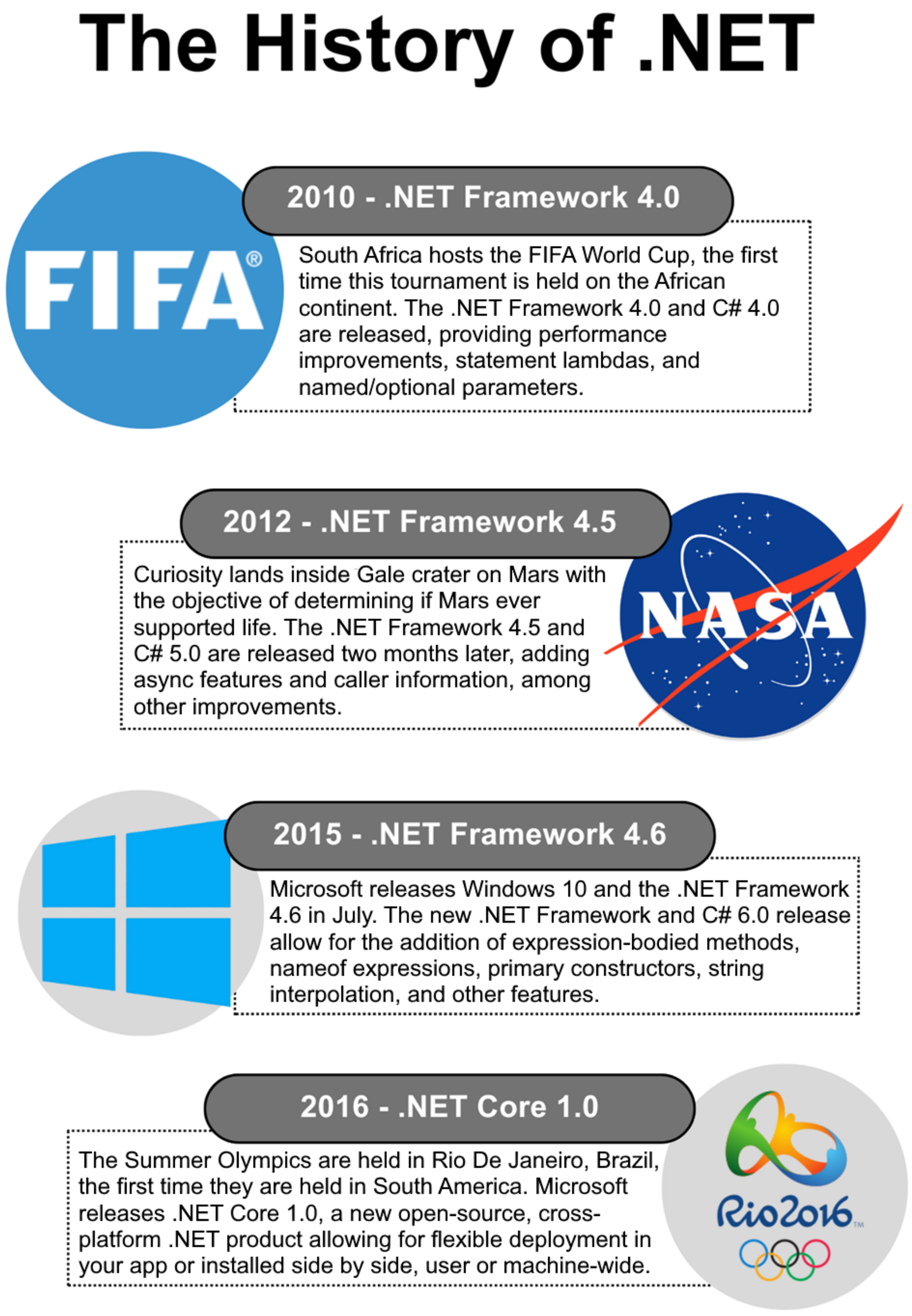
2006 - .NET Framework 3.0

Twitter is launched in March, allowing people to connect with each other on a different level. November sees the release of the .NET Framework 3.0 and C# 3.0 that includes WPF, WCF, WF, and Windows CardSpace.

2007 - .NET Framework 3.5

In June, Apple releases the iPhone and Microsoft releases the .NET Framework 3.5 later that year. The most notable feature is LINQ, adding native data querying capabilities to the .NET languages.

Figure 2: .NET 2002 to 2007



The History of .NET

2010 - .NET Framework 4.0

South Africa hosts the FIFA World Cup, the first time this tournament is held on the African continent. The .NET Framework 4.0 and C# 4.0 are released, providing performance improvements, statement lambdas, and named/optional parameters.

2012 - .NET Framework 4.5

Curiosity lands inside Gale crater on Mars with the objective of determining if Mars ever supported life. The .NET Framework 4.5 and C# 5.0 are released two months later, adding async features and caller information, among other improvements.

2015 - .NET Framework 4.6

Microsoft releases Windows 10 and the .NET Framework 4.6 in July. The new .NET Framework and C# 6.0 release allow for the addition of expression-bodied methods, nameof expressions, primary constructors, string interpolation, and other features.

2016 - .NET Core 1.0

The Summer Olympics are held in Rio De Janeiro, Brazil, the first time they are held in South America. Microsoft releases .NET Core 1.0, a new open-source, cross-platform .NET product allowing for flexible deployment in your app or installed side by side, user or machine-wide.

Figure 3: .NET 2010 to 2016

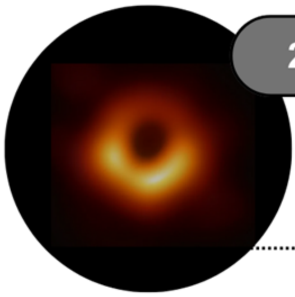
The History of .NET

2017 - .NET Framework 4.7

Nintendo releases the Switch worldwide, competing with the Xbox One and PlayStation 4. Microsoft releases the .NET Framework 4.7 as part of the Windows 10 Creators Update. The .NET Framework docs become available on docs.microsoft.com.

2017 - .NET Core 2.0

In August, .NET Core 2.0 and C# 7.0 are released, featuring out variables, tuples, discards, pattern matching, local functions, and throw expressions to name but a few.



2019 - .NET Framework 4.8

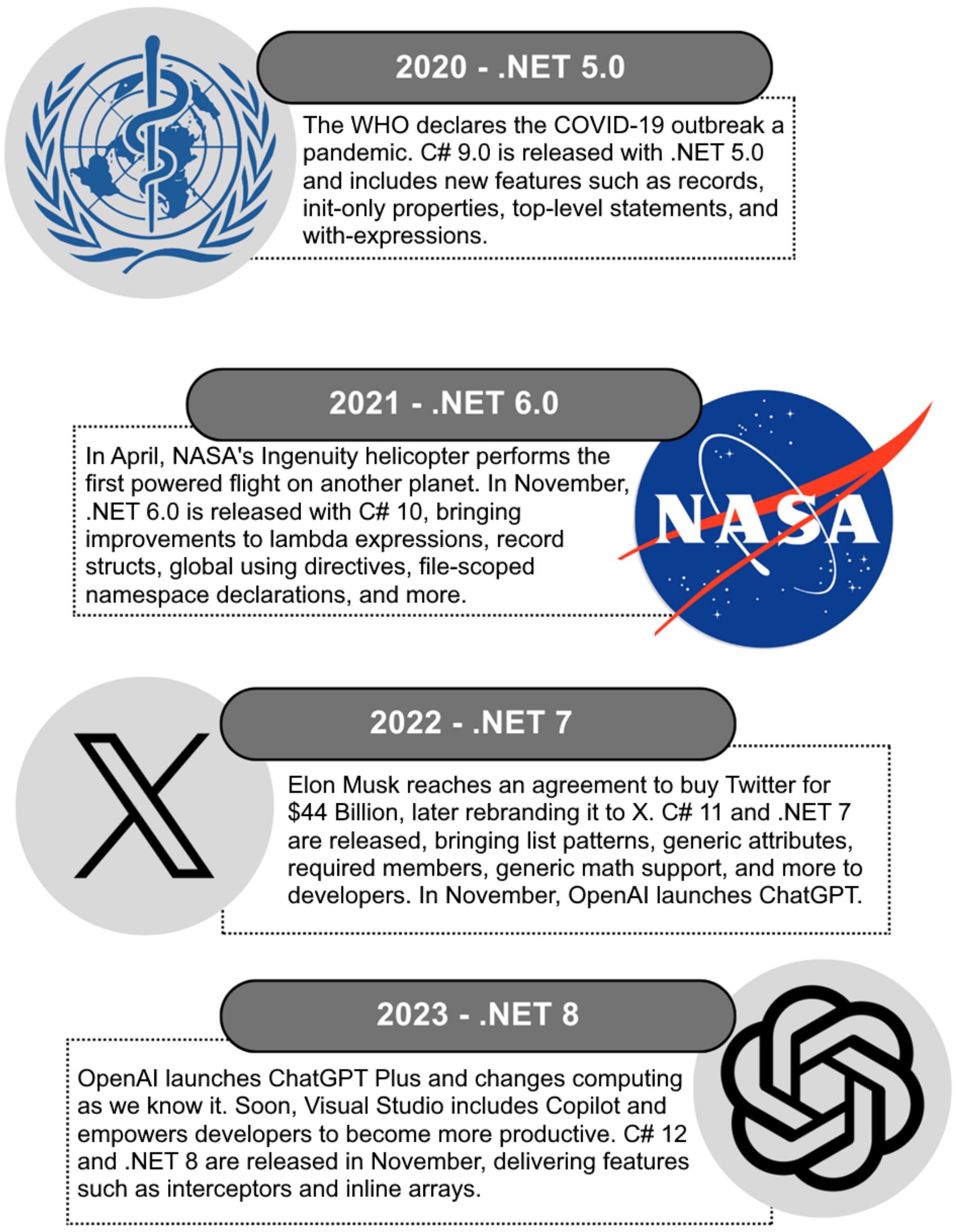
Scientists reveal the first ever image of a black hole. Microsoft releases .NET Framework 4.8 with improvements to JIT and NGEN, BCL, WinForms, WCF, and WPF.

2019 - .NET Core 3.0

In September, .NET Core 3.0 is released with C# 8.0, providing new features such as nullable reference types, using declarations, static local functions, and more.

Figure 4: .NET 2017 to 2019

The History of .NET



2020 - .NET 5.0

The WHO declares the COVID-19 outbreak a pandemic. C# 9.0 is released with .NET 5.0 and includes new features such as records, init-only properties, top-level statements, and with-expressions.

2021 - .NET 6.0

In April, NASA's Ingenuity helicopter performs the first powered flight on another planet. In November, .NET 6.0 is released with C# 10, bringing improvements to lambda expressions, record structs, global using directives, file-scoped namespace declarations, and more.

2022 - .NET 7

Elon Musk reaches an agreement to buy Twitter for $44 Billion, later rebranding it to X. C# 11 and .NET 7 are released, bringing list patterns, generic attributes, required members, generic math support, and more to developers. In November, OpenAI launches ChatGPT.

2023 - .NET 8

OpenAI launches ChatGPT Plus and changes computing as we know it. Soon, Visual Studio includes Copilot and empowers developers to become more productive. C# 12 and .NET 8 are released in November, delivering features such as interceptors and inline arrays.

Figure 5: .NET 2020 to 2023

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

The standard term support release of .NET 7.0 in November 2022 brought us closer to where we are today: with a unified .NET that supports cross-platform application development. It allows developers to create a host of different applications, from desktop apps to cloud apps, web applications, Unity games, mobile applications, and AI apps. With the release of MAUI, we can create cross-platform, native UI applications. The unification of .NET means that we now have a single BCL (or base class library) and SDK.

With the release of .NET 8 in November 2023, we now have a long-term support version of .NET until November 2026. For companies that never upgraded their software to .NET 7, take note that the end of support for .NET 6 was November 2024. If your company implements stringent change control processes, it would be prudent to start planning the implementation of .NET 8 across your applications early.

Let's have a look at the support windows for .NET next.

The support window

By now, most of your applications should be on .NET 6. However, I have come across a few applications myself that are still stuck on .NET Core 3.1—most notably, an institution in Johannesburg that is currently running their software, used to manage the compliance of financial professionals across South Africa, on .NET Core 3.1.

I use the word "stuck" carefully here. It's not that the company that developed the software has neglected to upgrade the version of .NET—the issue is that the company using the software refuses to incur additional cost to have the .NET version upgraded.

Note: You will incur some cost by upgrading your .NET version, since the software needs to be tested thoroughly after an upgrade to ensure the stability of the codebase.

If you are on the fence, however, see when your implemented .NET version support ends.

Table 1: Release and End-of-Support Dates

Version	Original Release	Release Type	End of Support
.NET 8	Nov. 14, 2023	LTS	Nov. 10, 2026
.NET 7	Nov. 8, 2022	STS	May 14, 2024
.NET 6	Nov. 8, 2021	LTS	Nov. 12, 2024

By looking at Table 1, you can determine when support for the version of .NET implemented by your software ends. If your software does not implement at least .NET 6, then you need to upgrade your software.

Table 2 outlines the version of .NET or .NET Core and when support ended.

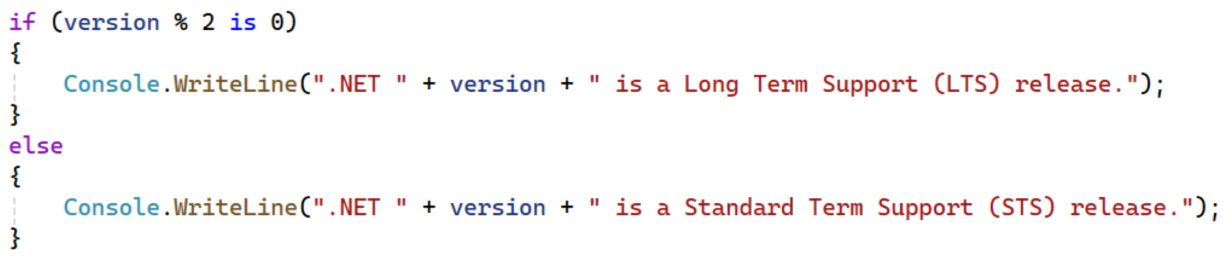
Table 2: Out of Support Versions

Version	Original Release	End of Support
.NET 5	Nov. 10, 2020	May 10, 2022
.NET Core 3.1	Dec. 3, 2019	Dec. 13, 2022
.NET Core 3.0	Sept. 23, 2019	Mar. 3, 2020
.NET Core 2.2	Dec. 4, 2018	Dec. 23, 2019
.NET Core 2.1	May 30, 2018	Aug. 21, 2021
.NET Core 2.0	Aug. 14, 2017	Oct. 1, 2018
.NET Core 1.1	Nov. 16, 2016	June 27, 2019
.NET Core 1.0	June 27, 2016	June 27, 2019

When we talk about long-term support (LTS) and standard-term support (STS), we are referring to two very distinct release cadences. The cadences are defined as follows:

- Long-term support: Supported for three years after initial release.
- Standard-term support: Supported for 18 months after initial release.

You will notice that odd-numbered releases are STS releases, and even-numbered releases are LTS releases. Therefore, you can also say that:



```
if (version % 2 is 0)
{
    Console.WriteLine(".NET " + version + " is a Long Term Support (LTS) release.");
}
else
{
    Console.WriteLine(".NET " + version + " is a Standard Term Support (STS) release.");
}
```

Figure 6: Cheeky Bit of Code

In all seriousness, the release cadence enables developers and businesses to plan ahead when it comes to their development roadmaps. To qualify for support, you need to install the latest patch update. This means that if you are running .NET 6, you must ensure that .NET 6.0.x is installed as a first step.

It is worth noting the following servicing policies, which are the same for LTS and STS releases.

Preview

Preview releases are usually offered to developers for testing. This is done ahead of the final release, but they are not supported by Microsoft.

Go-live

These releases are supported by Microsoft in production. Go-live releases are release candidate builds and are usually released just before the generally available (GA) releases.

Active support

During this support period, updates are provided to improve functionality and mitigate any security vulnerabilities. When it comes to functional improvements, these could typically include fixes to resolve reported crashes, performance issues, bugs, and adding support for new hardware platforms or operating system versions.

Maintenance support

At this stage, .NET releases are only updated to mitigate security vulnerabilities. The last six months of any release (LTS and STS) are considered the maintenance support period. Once the maintenance support period is over, the release is considered out of support.

End of life

When a release reaches end of life, Microsoft no longer provides fixes or updates for the release. If you use .NET versions that have reached end of life, your software and your data are at risk.

What's changed in .NET 8

.NET 8 will be supported for three years due to its long-term support release.

Note: You can download .NET 8 here.

This makes it worth your while to upgrade now. If you need a little more convincing, consider the following changes and improvements.

The .NET runtime

There have been many improvements made to performance (reminiscent of the .NET 7 release), garbage collection, and the core and extension libraries. I want to highlight just a few of them.

Garbage collection

The .NET 8 release includes the capability to adjust the memory limit on the fly. In cloud-service scenarios, demand can sometimes increase suddenly and then drop back down again. The cost effectiveness of these services is measured in how well they can scale up and down the resource consumption to meet the fluctuations on demand.

I would argue that scaling down is as important as scaling up as demand fluctuates. This means that when a decrease in demand is detected, a resource can scale down its consumption by reducing its memory limit.

Before .NET 8, however, this scaling down would fail because the garbage collector was not aware of any changes and might allocate more memory than the new limit. By calling **`GC.RefreshMemoryLimit()`**, you can update the garbage collection to the new limit.

There are, however, some limitations of which to take note:

- On 32-bit platforms, a new heap hard limit can’t be established by .NET if there isn’t already one.
- The **`GC.RefreshMemoryLimit()`** call might fail and return a non-zero status code. This happens because the scale-down is considered too aggressive and does not leave enough memory for the garbage collection to work with. In this case, calling **`GC.Collect(2, GCCollectionMode.Aggressive)`** might solve the issue by shrinking the current memory usage, allowing you to call **`GC.RefreshMemoryLimit()`** again.
- Scaling up the memory limit beyond what the garbage collector believes the startup process can handle allows the **`GC.RefreshMemoryLimit()`** to succeed, but it will not use more memory than what the garbage collector perceives as the limit.

The following code in Code Listing 1 illustrates setting the heap hard limit to 100 MiB (mebibytes).

Code Listing 1: Setting the Heap Hard Limit

```
internal class Program
{
    static void Main(string[] args)
    {
        AppContext.SetData("GCHeapHardLimit", (ulong)100 * 1024 * 1024);
        GC.RefreshMemoryLimit();
    }
}
```

If this hard limit is invalid, the **`GC.RefreshMemoryLimit()`** will throw an **`InvalidOperationException`**. An invalid hard limit can be, for example, when the hard limit that will be set by the refresh is lower than what's already committed.

Core .NET libraries

There are many improvements and additions in the core .NET libraries. Some of these improvements were made to the following features.

Serialization

A few serialization and deserialization improvements have been made to **`System.Text.Json`** in .NET 8. I will not discuss all of them, but here are some of the more interesting improvements and additions.

Handling missing members

One of the features that I am quite excited about is the ability to handle missing members during deserialization.

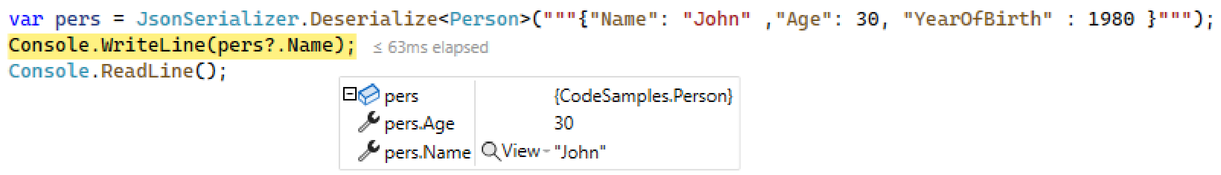
Usually, if you receive a JSON payload, you deserialize this into a POCO (plain old CLR object). If your JSON payload contains properties that are missing in your POCO when you deserialize the JSON, they are just ignored.

.NET 8 allows you to specify that all members must exist in the payload; otherwise, a **`JsonException`** will be thrown. Consider Code Listing 2.

Code Listing 2: Deserializing POCO with Missing Property

```
internal class Program
{
    static void Main(string[] args)
    {
        var pers = JsonSerializer.Deserialize<Person>("""{"Name": "John"
,"Age": 44, "YearOfBirth" : 1980 }""");
        Console.WriteLine(pers?.Name);
        Console.ReadLine();
    }
}

public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

Here, we are deserializing a JSON payload that includes a value for **`YearOfBirth`** that the **`Person`** POCO does not contain.



```
var pers = JsonSerializer.Deserialize<Person>("""{"Name": "John" ,"Age": 30, "YearOfBirth" : 1980 }""");
Console.WriteLine(pers?.Name);  ≤ 63ms elapsed
Console.ReadLine();
```

```
pers        {CodeSamples.Person}
pers.Age    30
pers.Name   View "John"
```

Figure 7: Deserialization Results of Person

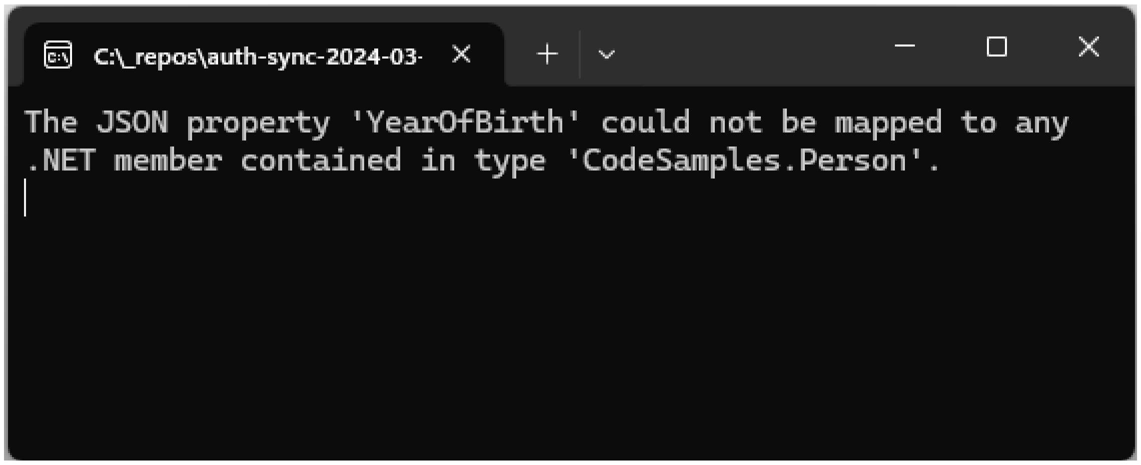
After deserialization, we can see that the **`Person`** class only contains values for **`Age`** and **`Name`**, because these are the only properties it contained. But our payload contained an additional value for **`YearOfBirth`**, which was just ignored.

Now consider Code Listing 3.

Code Listing 3: Annotating the POCO

```
internal class Program
{
    static void Main(string[] args)
    {
        try
        {
            var pers = JsonSerializer.Deserialize<Person>("""{"Name":
"John" ,"Age": 30, "YearOfBirth" : 1980 }""");
            Console.WriteLine(pers?.Name);
        }
        catch (JsonException ex)
        {
            Console.WriteLine(ex.Message);
        }
        Console.ReadLine();
    }
}

[JsonUnmappedMemberHandling(JsonUnmappedMemberHandling.Disallow)]
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

Notice that I have added the **`JsonUnmappedMemberHandling`** attribute and specified that it should be disallowed.



```
The JSON property 'YearOfBirth' could not be mapped to any
.NET member contained in type 'CodeSamples.Person'.
```

Figure 8: JsonUnmappedMemberHandling Result

If we run the **`CodeSamples`** console application a second time, the **`JsonException`** will be thrown and handled by the **`try catch`**, as seen in Code Listing 3.

The other option is to set the attribute to **`JsonUnmappedMemberHandling.Skip`**, but this is the default behavior anyway, so it wouldn't make much sense to add it (unless you want to be explicit in your code).

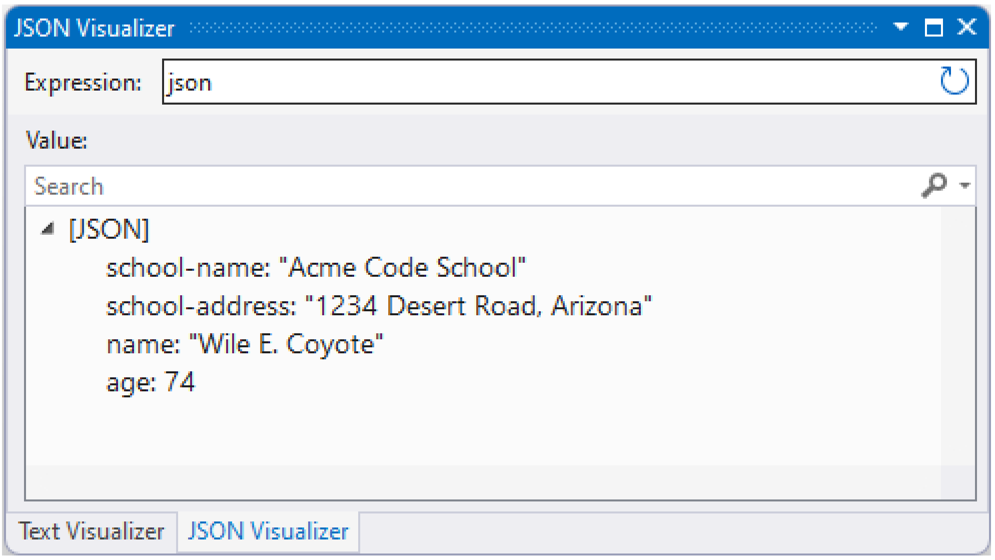
Naming policies

I don't think I can love a feature more than I love this feature: new naming policies for the **`JsonNamingPolicy`** class. These are (I kid you not) **`snake_case`** and **`kebab-case`** property name conversions. Consider the code in Code Listing 4.

Code Listing 4: JSON Naming Policies

```
internal class Program
{
    static void Main(string[] args)
    {
        Student student = new()
        {
            Name = "Wile E. Coyote",
            Age = 74,
            SchoolName = "Acme Code School",
            SchoolAddress = "1234 Desert Road, Arizona"
        };

        var options = new JsonSerializerOptions
        {
            PropertyNamingPolicy = JsonNamingPolicy.KebabCaseLower,
        };
        string json = JsonSerializer.Serialize(student, options);
    }
}

public class Student : Person
{
    public required string SchoolName { get; set; }
    public required string SchoolAddress { get; set; }
}

[JsonUnmappedMemberHandling(JsonUnmappedMemberHandling.Disallow)]
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

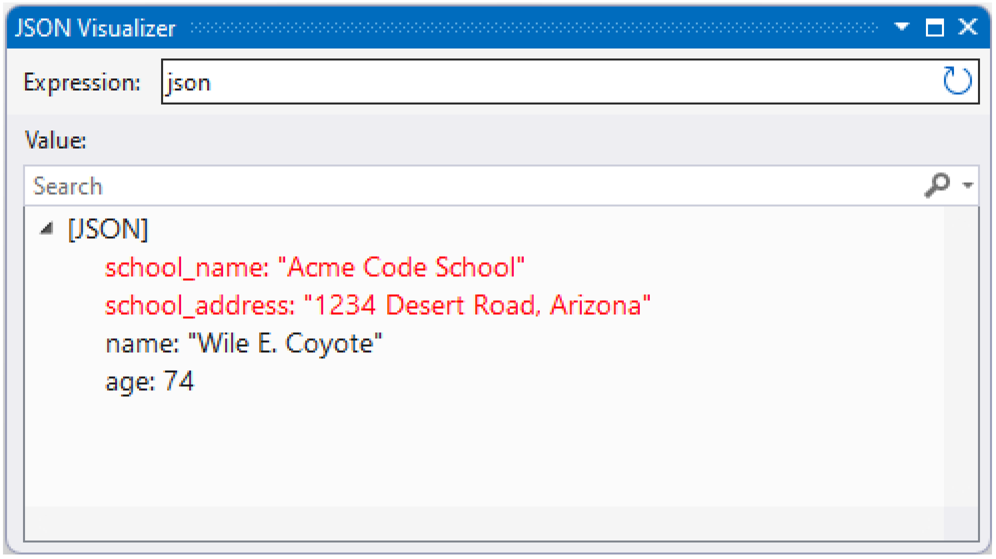
You will notice that we specify **`KebabCaseLower`** in the **`JsonSerializerOptions`**. Running the code and inspecting the resultant JSON, you will see that **`SchoolName`** and **`SchoolAddress`** have been kebab-cased, as seen in Figure 9.



```
[JSON]
    school-name: "Acme Code School"
    school-address: "1234 Desert Road, Arizona"
    name: "Wile E. Coyote"
    age: 74
```

Figure 9: Kebab Case

Go ahead and change the **`JsonNamingPolicy`** from **`KebabCaseLower`** to **`SnakeCaseLower`** and run the code again.



```
[JSON]
    school_name: "Acme Code School"
    school_address: "1234 Desert Road, Arizona"
    name: "Wile E. Coyote"
    age: 74
```

Figure 10: Snake Case

You will see that snake casing has been applied to **`SchoolName`** and **`SchoolAddress`**. The following naming policies are available for the **`JsonNamingPolicy`** class:

- CamelCase
- KebabCaseLower
- KebabCaseUpper
- SnakeCaseLower
- SnakeCaseUpper

I love the flexibility that this offers developers when deciding on a JSON naming policy.

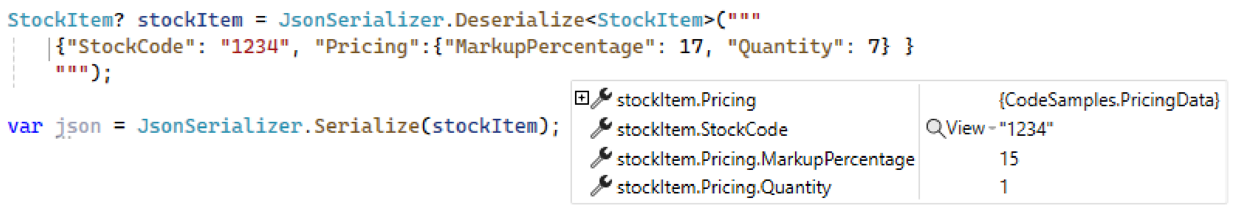
Read-only properties

Deserializing into read-only properties is now possible. Consider the code in Code Listing 5. I have a **`StockItem`** class with a read-only property for **`Pricing`**. The property defaults to **`15`** percent and a **`Quantity`** of **`1`**.

Code Listing 5: Deserializing into Read-Only Properties in .NET 7

```
internal class Program
{
    static void Main(string[] args)
    {
        StockItem? stockItem = JsonSerializer.Deserialize<StockItem>("""
            {"StockCode": "1234", "Pricing":{"MarkupPercentage": 17,
"Quantity": 7} }
            """);

        var json = JsonSerializer.Serialize(stockItem);
    }
}

public class StockItem
{
    public required string StockCode { get; set; }

    public PricingData Pricing { get; } = new()
    {
        MarkupPercentage = 15,
        Quantity = 1
    };
}

public class PricingData
{
    public required int MarkupPercentage { get; set; }
    public required int Quantity { get; set; }
}
```

Running the code using .NET 7, the JSON payload deserialized into the **`StockItem`** class specifies a **`MarkupPercentage`** of **`17`** percent and a **`Quantity`** of **`7`**.



```
StockItem? stockItem = JsonSerializer.Deserialize<StockItem>("""
    {"StockCode": "1234", "Pricing":{"MarkupPercentage": 17, "Quantity": 7} }
    """);

var json = JsonSerializer.Serialize(stockItem);
```

```
stockItem.Pricing                      {CodeSamples.PricingData}
stockItem.StockCode                    "1234"
stockItem.Pricing.MarkupPercentage     15
stockItem.Pricing.Quantity             1
```

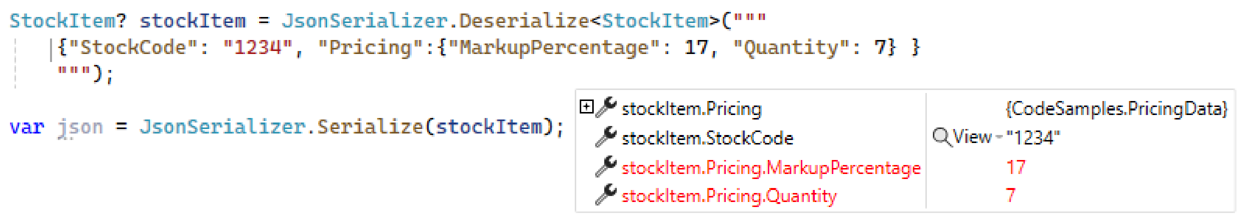
Figure 11: Results from Setting Read-Only Properties in .NET 7

Looking at the results in Figure 11, you will notice that the deserialized JSON produces a **`StockItem`** object that ignored the **`17`** percent and **`Quantity`** of **`7`** set in the JSON. In .NET 8, however, we can tell it to populate the read-only property **`MarkupPercentage`** by setting the **`JsonObjectCreationHandling`** attribute to **`Populate`**, as seen in Code Listing 6.

Code Listing 6: Modifying the StockItem Class

```
[JsonObjectCreationHandling(JsonObjectCreationHandling.Populate)]
public class StockItem
{
    public required string StockCode { get; set; }

    public PricingData Pricing { get; } = new()
    {
        MarkupPercentage = 15,
        Quantity = 1
    };
}
```

Go ahead and run the code again and take a look at the **`stockItem`** variable, as seen in Figure 12.



```
StockItem? stockItem = JsonSerializer.Deserialize<StockItem>("""
    {"StockCode": "1234", "Pricing":{"MarkupPercentage": 17, "Quantity": 7} }
    """);

var json = JsonSerializer.Serialize(stockItem);
```

```
stockItem.Pricing                      {CodeSamples.PricingData}
stockItem.StockCode                    View - "1234"
stockItem.Pricing.MarkupPercentage     17
stockItem.Pricing.Quantity             7
```

Figure 12: The Read-Only Properties Have Been Set

Looking at the variable, you will notice that the **`MarkupPercentage`** and **`Quantity`** values have been set to what was contained in the JSON payload. I have to admit that I am not sure whether I like this behavior.

You can read more about populating initialized properties here.

Time abstraction

With .NET 8, we get a new time provider class, **`System.TimeProvider`**, and an **`ITimer`** interface, **`System.Threading.ITimer`**. Let's see how to use them and how to test time-dependent code. I have a service called **`TaskRunnerService`** that simply returns the duration of the elapsed time, as seen in Code Listing 7.

Code Listing 7: The TaskRunnerService Class

```
public class TaskRunnerService
{
    private readonly TimeProvider _timeProvider;
    public TaskRunnerService(TimeProvider timeProvider) => _timeProvider
= timeProvider;


    public TimeSpan MorningTaskRunner()
    {
        var startTime = _timeProvider.GetTimestamp();
        Task.Delay(5000).Wait();
        var endTime = _timeProvider.GetTimestamp();

        var duration = _timeProvider.GetElapsedTime(startTime, endTime);
        return duration;
    }
}
```

Next, I have a service called **`TimeService`** that checks to see if the current time is in the morning, as seen in Code Listing 8.

Code Listing 8: The TimeService Class

```
public class TimeService
{
    private readonly TimeProvider _timeProvider;

    public TimeService(TimeProvider timeProvider) => _timeProvider = 
timeProvider;

    public bool RunMorningTasks()
    {
        var currentTime = _timeProvider.GetLocalNow();
        return currentTime.Hour <= 11;
    }
}
```

I then created an API that checks if the current time is in the morning, and if so, it runs the **`TaskRunnerService`**. To use these services, I need to add them to the **`Program`** class of the API, as seen in Code Listing 9. I also want to use the new **`TimeProvider`** class, so I need to add this service, too.

Code Listing 9: The Program Class

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

        // Add services to the container.
        builder.Services.AddSingleton(TimeProvider.System);
        builder.Services.AddSingleton<TimeService>();
        builder.Services.AddSingleton<TaskRunnerService>();

        builder.Services.AddControllers();
        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.MapControllers();

        app.Run();
    }
}
```

My **`TimeController`** contains the **`GET`** endpoint **`RunMorningTasks`** that does all the work. You will see this code in Code Listing 10. First, the API checks to see if it is morning, and if so, runs the **`TaskRunnerService`**.

Code Listing 10: The RunMorningTasks API

```
[ApiController]
[Route("api/[action]")]
public class TimeController : Controller
{
    private readonly TimeService _timeService;
    private readonly TaskRunnerService _taskRunnerService;

    public TimeController(TimeService timeService, TaskRunnerService
taskRunnerService)
    {
        _timeService = timeService;
        _taskRunnerService = taskRunnerService;
    }


    [HttpGet]
    [ActionName("RunMorningTasks")]
    public IActionResult Get()
    {

        var isMorning = _timeService.RunMorningTasks();
        if(isMorning)
        {
            var duration = _taskRunnerService.MorningTaskRunner();
            return Ok($"Ran morning tasks - duration {duration}");
        }

        return Ok("Morning tasks not scheduled to run");
    }
}
```

Looking back to the code for the **TaskRunnerService** in Code Listing 7, you will notice that I use the new **TimeProvider** class to get the duration of the running task.

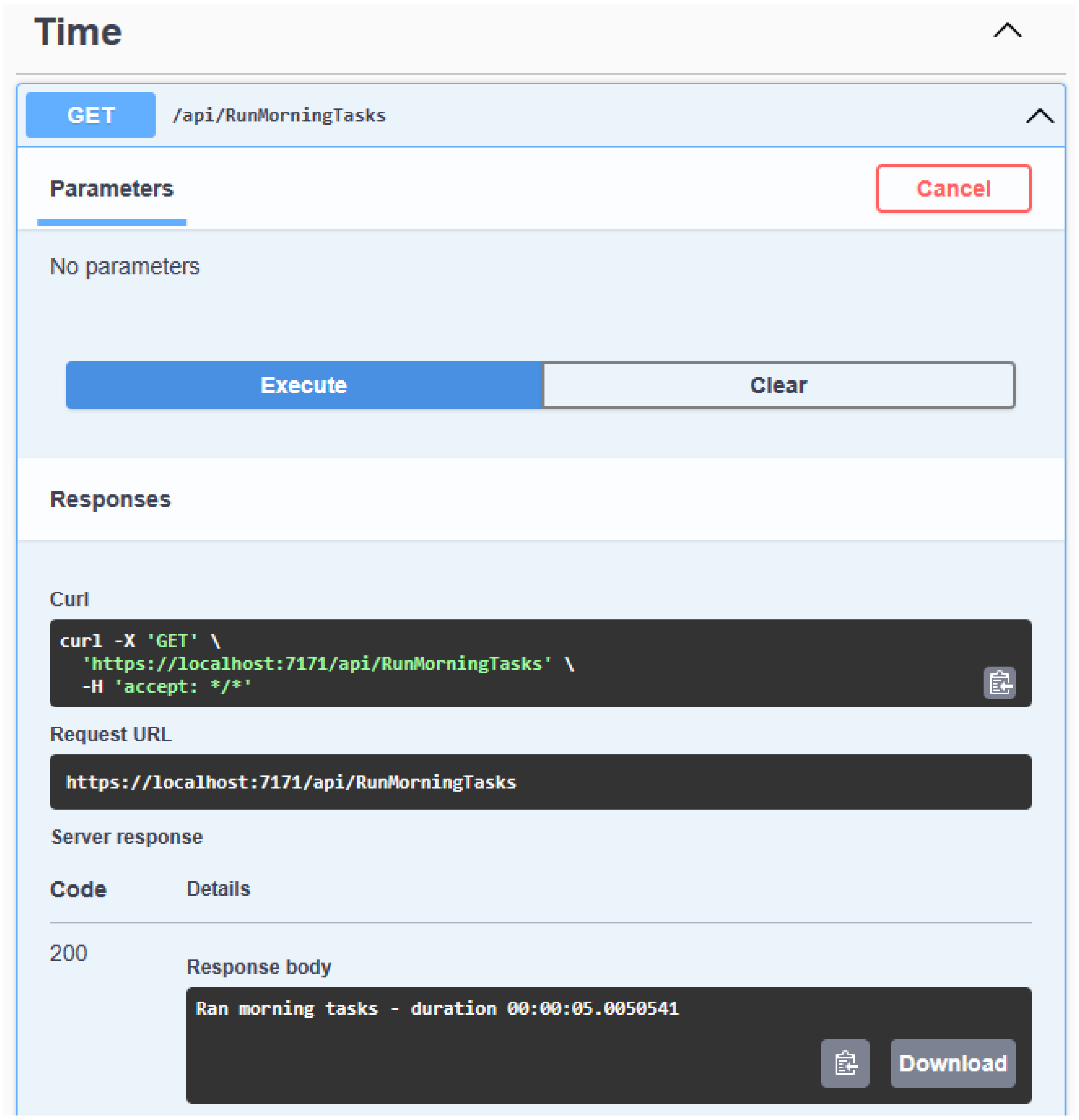
Note: Imagine replacing *Task.Delay(5000).Wait()* ***in Code Listing 7 with a long-running task.***

The **TaskRunnerService** uses **GetTimeStamp**, provided by the **TimeProvider**, to give you a system-based, high-frequency time stamp designed for small time-interval measurements with high accuracy.

Using **TimeProvider** also gives you access to **GetElapsedTime**, which calculates the difference between the start and end times very precisely in a high-frequency scenario.

The reason for this high level of accuracy is because **GetTimeStamp** actually works with your machine to return the exact time stamp. This is much more accurate than what the **DateTime** class can provide.

Calling the API, as seen in Figure 13, will run the **TaskRunnerService**, because it is still morning where I live.



```
curl -X 'GET' \
  'https://localhost:7171/api/RunMorningTasks' \
  -H 'accept: */*'
```

```
https://localhost:7171/api/RunMorningTasks
```

```
Ran morning tasks - duration 00:00:05.0050541
```

Figure 13: Calling RunMorningTasks API

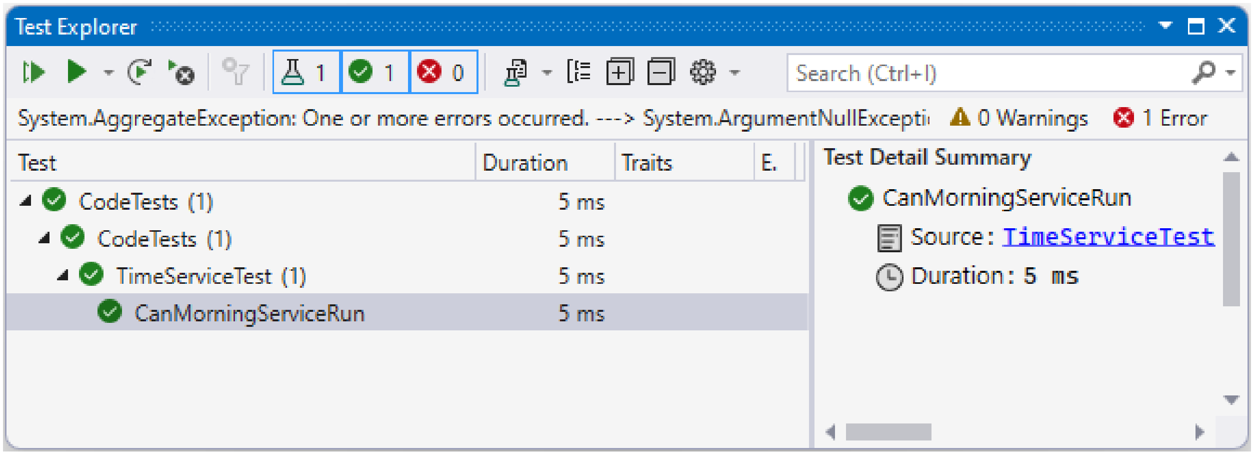
Let us think about unit testing our **`TimeService`**. The **`TimeProvider`** class comes in handy here, too. Consider the unit test seen in Code Listing 11. In it, I am able to override the **`GetUtcNow`** method, as well as override the **`LocalTimeZone`** property. I kept the overridden **`GetUtcNow`** method as is (using **`UtcNow`**), but you can supply hard-coded values instead of specifying **`_utc.Year`**, **`_utc.Month`**, and so on.

What I did do, however, was override the **`LocalTimeZone`** property with a new time zone.

Code Listing 11: The TimeProvider Unit Test

```
namespace CodeTests
{
    [TestClass]
    public class TimeServiceTest
    {
        public class TimeProviderMorning : TimeProvider
        {
            private readonly DateTimeOffset _utc;

            public TimeProviderMorning() => _utc = DateTimeOffset.UtcNow;

            public override DateTimeOffset GetUtcNow() => new(_utc.Year,
_utc.Month, _utc.Day, _utc.Hour, _utc.Minute, _utc.Second,
TimeSpan.Zero);

            public override TimeZoneInfo LocalTimeZone =>
TimeZoneInfo.FindSystemTimeZoneById("Pacific Standard Time");
        }

        [TestMethod]
        public void CanMorningServiceRun()
        {
            var timeProviderMorning = new TimeProviderMorning();
            var timeService = new TimeService(timeProviderMorning);
            var canRunMorningTasks = timeService.RunMorningTasks();

            Assert.IsTrue(canRunMorningTasks);
        }
    }
}
```

Setting the time zone to PST will return **`true`** when running the test, as seen in Figure 14. It is currently very early morning in Los Angeles.


Test Explorer

System.AggregateException: One or more errors occurred. ---> System.ArgumentNullExcepti 0 Warnings 1 Error

```
Test                              Duration
CodeTests (1)                     5 ms
  CodeTests (1)                   5 ms
    TimeServiceTest (1)           5 ms
      CanMorningServiceRun        5 ms
```

Test Detail Summary

CanMorningServiceRun
Source: TimeServiceTest
Duration: 5 ms

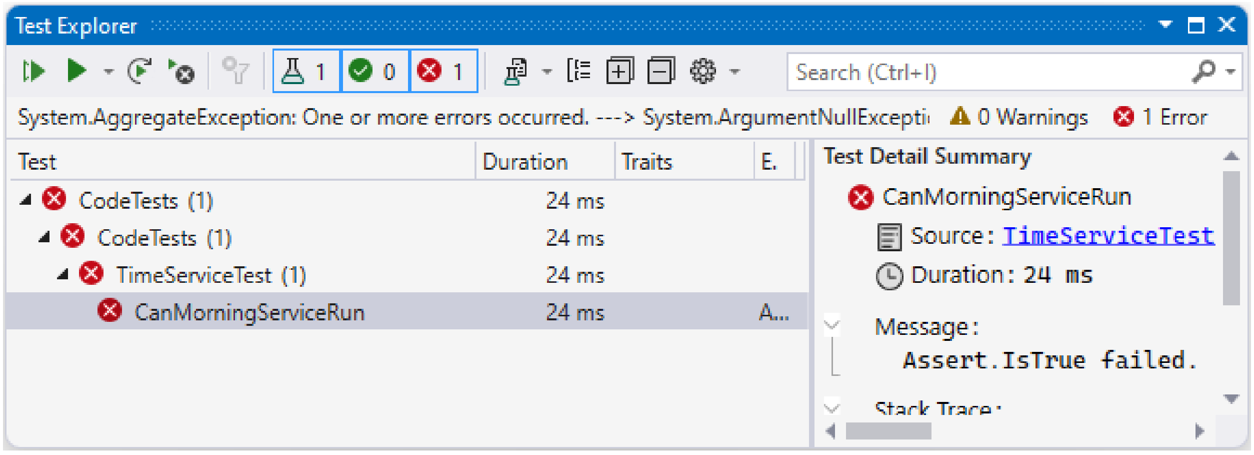
Figure 14: Passing Test with PST

Changing the time zone to CST paints a different picture when it comes to our unit test. Modify the **`TimeProviderMorning`** class, as seen in Code Listing 12, by specifying the time zone as China Standard Time.

Code Listing 12: Modified TimeProviderMorning Class

```
public class TimeProviderMorning : TimeProvider
{
    private readonly DateTimeOffset _utc;

    public TimeProviderMorning() => _utc = DateTimeOffset.UtcNow;

    public override DateTimeOffset GetUtcNow() => new(_utc.Year,
_utc.Month, _utc.Day, _utc.Hour, _utc.Minute, _utc.Second,
TimeSpan.Zero);

    public override TimeZoneInfo LocalTimeZone =>
TimeZoneInfo.FindSystemTimeZoneById("China Standard Time");
}
```

Running the unit test again results in a failed test, as seen in Figure 15.


Test Explorer

System.AggregateException: One or more errors occurred. ---> System.ArgumentNullExcepti   0 Warnings   1 Error

Test	Duration	Traits	E.
CodeTests (1)	24 ms		
CodeTests (1)	24 ms		
TimeServiceTest (1)	24 ms		
CanMorningServiceRun	24 ms		A...

Test Detail Summary

CanMorningServiceRun

Source: TimeServiceTest

Duration: 24 ms

Message:
  Assert.IsTrue failed.

Stack Trace:

Figure 15: Failing Test Results

In fact, if you debug the unit test and placed a break point on the return statement, you will see that the current time is UTC+08:00, which is China Standard Time.

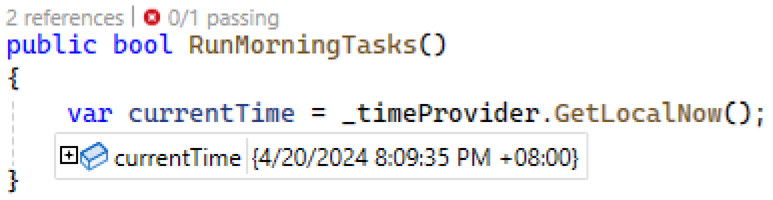

```
2 references | 0/1 passing
public bool RunMorningTasks()
{
    var currentTime = _timeProvider.GetLocalNow();
      currentTime {4/20/2024 8:09:35 PM +08:00}
}
```

Figure 16: Debugging the RunMorningTasks Method

It's always a challenge to handle date and time in software development, especially when dealing with multiple time zones. Sprinkle a little bit of daylight savings time in this mix, and you have a tough cookie to chew on.

Using the new **TimeProvider** class isn't a magic bullet. It is, however, a welcome new feature to .NET, helping developers deal with time- and date-dependent code, especially when using unit tests.

Randomness

With the release of .NET 8, Microsoft introduced two brand-new methods on the **Random** class that are extremely fast and memory-efficient. First, let's have a look at the **Shuffle** method. As seen in Code Listing 13, we are passing an array of strings to the **Shuffle** method of the **Random** class.

Code Listing 13: The Shuffle Method

```
var cities = new[]
{
    "Raleigh", "Tampa", "Los Angeles", "New York", "Chicago"
};
```

```
Random.Shared.Shuffle(cities);
foreach (var c in cities)
{
    Console.WriteLine(c);
}
```

As can be expected, this will shuffle the cities in the array and output them in a random order. Taking a peek at the **`Shuffle`** method, you will notice the following, as seen in Code Listing 14.

Code Listing 14: Peeking Inside the Shuffle Method

```
public void Shuffle<T>(T[] values)
{
    ArgumentNullException.ThrowIfNull(values);
    Shuffle(values.AsSpan());
}

public void Shuffle<T>(Span<T> values)
{
    int n = values.Length;

    for (int i = 0; i < n - 1; i++)
    {
        int j = Next(i, n);

        if (j != i)
        {
            T temp = values[i];
            values[i] = values[j];
            values[j] = temp;
        }
    }
}
```

The **`Shuffle`** method accepts an array or a **`Span`**. If you pass it an array, it will be changed to a **`Span`** anyway, before being shuffled. This makes it extremely efficient, as it performs an in-place shuffle of the respective data structures, instead of returning a new array or **`Span`**.

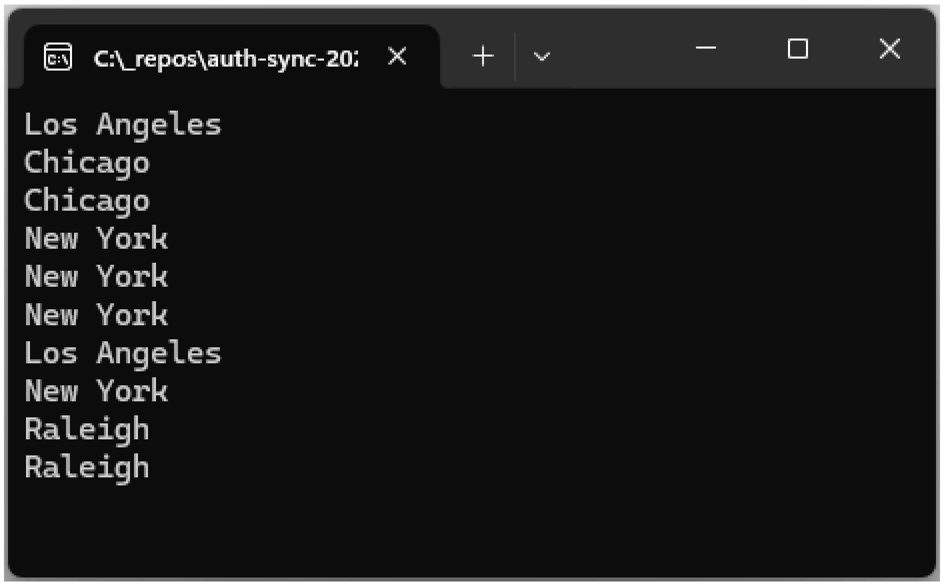
Therefore, keep this in mind if you have created code that shuffles large arrays, as this might give you a performance boost.

The second new method we will look at is the **`GetItems`** method. Take a look at the code in Code Listing 15. Here, we are telling .NET to use our **`cities`** array and return to us a random array of length **`10`**, containing the city names in our **`cities`** array.

Code Listing 15: Using GetItems

```
var cities = new[]
{
    "Raleigh", "Tampa", "Los Angeles", "New York", "Chicago"
};

var generatedCities = Random.Shared.GetItems<string>(cities, 10);

foreach (var c in generatedCities)
{
    Console.WriteLine(c);
}
```

Running the code in Code Listing 15 results in the output seen in Figure 17.



```
Los Angeles
Chicago
Chicago
New York
New York
New York
Los Angeles
New York
Raleigh
Raleigh
```

Figure 17: The GetItems Result

The **`GetItems`** method does not create any new city names but merely returns to us an array with the names in our **`cities`** array.

This feature might not be useful for most applications, but it can be very useful in specialized systems, such as those in machine learning.

Lastly, I want to have a look at the **`RandomNumberGenerator`** class. You will notice that this class also gets the **`Shuffle`** and **`GetItems`** methods and does the same thing as the **`Random`** class, but with cryptographically secure randomness.

More interestingly, the **RandomNumberGenerator** class has two new methods, called **GetHexString** and **GetString**, as seen in Code Listing 16.

The **GetHexString** method will randomly generate a cryptographically secure string of 25 characters in length, which you can use to create unpredictable IDs (for example).

The **GetString** method accepts a string of characters and a length and returns a cryptographically secure random string of characters (based on the characters you provided as parameter) of the length you specified.

Code Listing 16: The RandomNumberGenerator Class

```
var hexStr = RandomNumberGenerator.GetHexString(25);
// 9CDCD0C99A7E471DE5F4D2E0F

var genString =
RandomNumberGenerator.GetString("bcdefgBCDEFG0123456789=><!@#$%^*()-+&",
15);
// G#7cg-g9g5g=b16
```

A word of caution, though: while this can be very useful in many areas of your code, do not rely solely on this for generating passwords. While the returned random string is generated in a cryptographically secure manner, it is very easy to generate a weak password. This is especially true if the length you provide is small, and the characters you provide do not strike a nice balance among lowercase, uppercase, and special characters.

Performance-focused types

As with .NET 7, Microsoft has focused on performance with the release of .NET 8. There are some nice improvements out of the box, as well as some new types that you can use in your code. A little later on, we will look at benchmarking some of this code. For now, let’s just have a look at the new types.

FrozenSet and FrozenDictionary

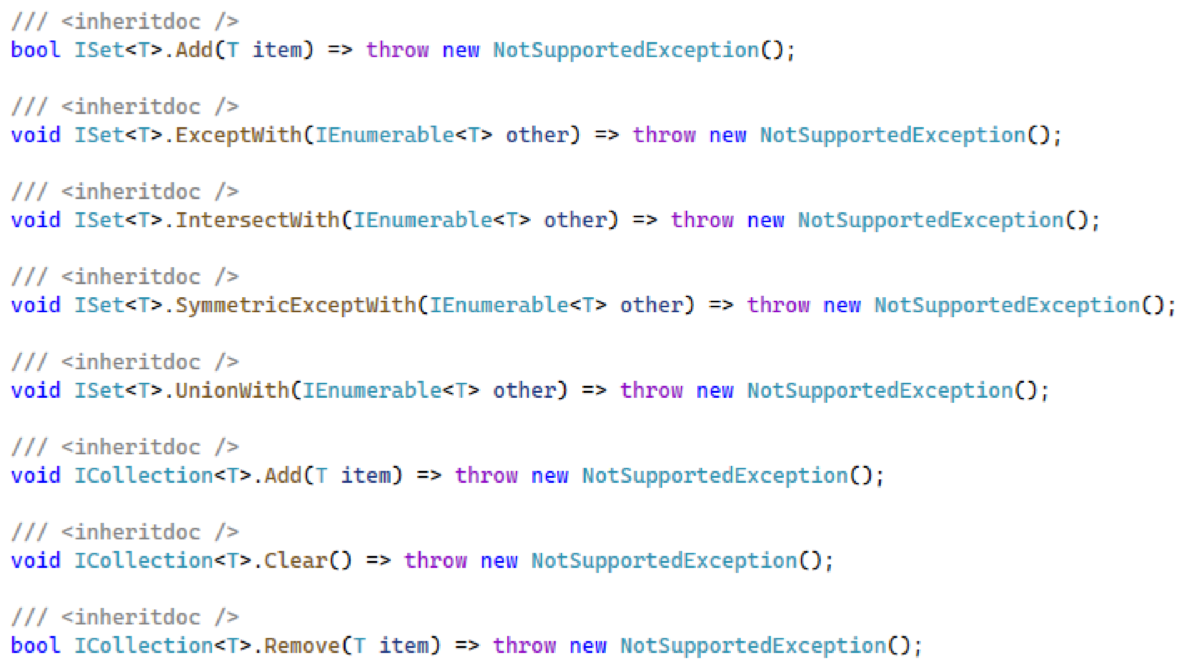
In the .NET 8 System.Collections.Frozen namespace, we now have two new collection types: **FrozenDictionary<TKey, Tval>** and **FrozenSet<T>**. Once created, you cannot make changes to any keys or values. As you can imagine, this leads to faster read operations using **TryGetValue()**, for example. With a long-lived service, types like these are really useful because they are populated on first use and persisted for the service duration.

Code Listing 17: Using the FrozenSet

```
var lst = new List<string> { "John", "Mark", "Jane", "Jeremy", "Daton",
"Sally", "Mary" };

var frozenSet = lst.ToFrozenSet();
var willTheRealJeremyPleaseStandup = frozenSet.Contains("Jeremy");
```

Taking a look at the **ToFrozenSet** method in Code Listing 17, for example, the code is not particularly mind-blowing in and of itself. You will get what you expect here, which is a frozen set containing the names of the people in the list provided.

What I do find interesting is what's underneath. As mentioned before, this frozen set cannot be modified. You can see this behavior if you look at a subset of the code contained in the **FrozenSet** class, as seen in Figure 18. The **FrozenSet** class implements the **ISet** Interface.

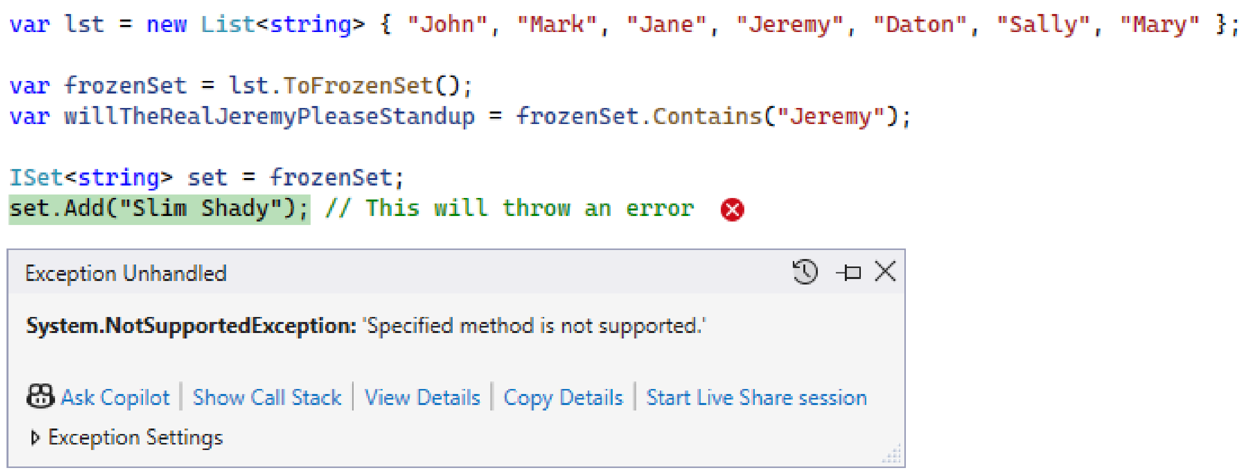


```
/// <inheritdoc />
bool ISet<T>.Add(T item) => throw new NotSupportedException();

/// <inheritdoc />
void ISet<T>.ExceptWith(IEnumerable<T> other) => throw new NotSupportedException();

/// <inheritdoc />
void ISet<T>.IntersectWith(IEnumerable<T> other) => throw new NotSupportedException();

/// <inheritdoc />
void ISet<T>.SymmetricExceptWith(IEnumerable<T> other) => throw new NotSupportedException();

/// <inheritdoc />
void ISet<T>.UnionWith(IEnumerable<T> other) => throw new NotSupportedException();

/// <inheritdoc />
void ICollection<T>.Add(T item) => throw new NotSupportedException();

/// <inheritdoc />
void ICollection<T>.Clear() => throw new NotSupportedException();

/// <inheritdoc />
bool ICollection<T>.Remove(T item) => throw new NotSupportedException();
```

Figure 18: The Unsupported ISet Implementations

The **FrozenSet** does not, however, support the methods of **ISet** that mutate data. Because **FrozenSet** implements the **ISet** interface, and because **ISet** defines the number of methods a set should have, the **FrozenSet** must provide an implementation for all the methods of **ISet**, even the unsupported methods. Because the **FrozenSet** is designed to be immutable, it simply throws a **NotSupportedException** in the implementation.

You can even test this by casting the instance of the **FrozenSet** to an **ISet<string>**, as seen in Figure 19, and trying to add some data. Slim Shady will remain seated and hidden, because trying to add to the set throws the exception.



```
var lst = new List<string> { "John", "Mark", "Jane", "Jeremy", "Daton", "Sally", "Mary" };

var frozenSet = lst.ToFrozenSet();
var willTheRealJeremyPleaseStandup = frozenSet.Contains("Jeremy");

ISet<string> set = frozenSet;
set.Add("Slim Shady"); // This will throw an error
```

```
Exception Unhandled

System.NotSupportedException: 'Specified method is not supported.'
```

Ask Copilot | Show Call Stack | View Details | Copy Details | Start Live Share session
Exception Settings

Figure 19: Mutating Will Throw an Exception

You might be wondering why the **`FrozenSet`** class even implements **`ISet`** if it doesn't support all the operations of a set. This is because **`ISet<T>`** also contains a number of useful methods for querying a set, such as **`Contains`**. Therefore, **`FrozenSet`** implements **`ISet`** so that it can provide implementations for these methods.

The **`FrozenDictionary`** works in exactly the same manner, as seen in Code Listing 18.

Code Listing 18: The FrozenDictionary

```
var dictionary = new Dictionary<string, int>
{
    { "John", 38 },
    { "Mark", 42 },
    { "Jane", 24 },
};

var frozenDictionary = dictionary.ToFrozenDictionary();
```

This time, the **`FrozenDictionary`** implements the **`IDictionary`** interface. Similarly, it needs to add the implementations of the interface it implements and does so by throwing a **`NotSupportedException`** for all the methods that would mutate the data.

SearchValue<T>

.NET 8 now also includes a new **`SearchValues<T>`** type. Methods such as **`MemoryExtensions.ContainsAny`**, which checks to see if a value is contained in a collection, have been modified to add new overloads to accept an instance of this new type.

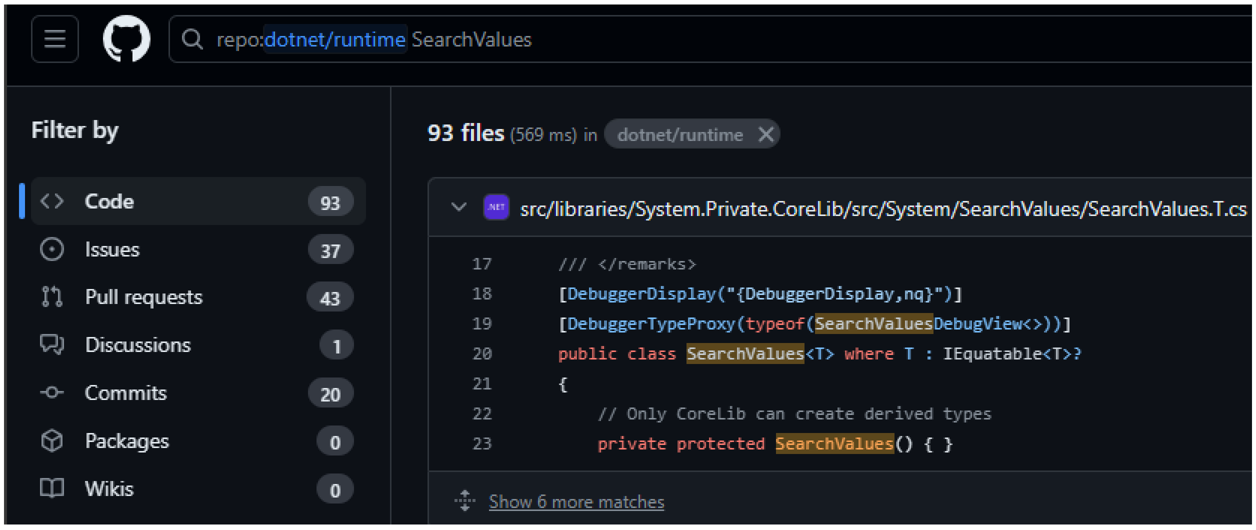
This means that you can now do what you see in Code Listing 19. I have created an instance of **`SearchValues<char>`**, but you can also have an instance of **`SearchValues<byte>`**.

Code Listing 19: The New SearchValues Type

```
var strSear = @"<>:""/\|?*";
SearchValues<char> illegalChars = SearchValues.Create(strSear);

var fileName = @"Important_|_File.txt";

var illegal = fileName.AsSpan().ContainsAny(illegalChars);
// illegal = true
```

You might be wondering why this new type matters. Well, as it turns out, when you create an instance of this type, everything that is required to optimize subsequent searches is derived. Having all this work done up front means a performance boost for your code.

In fact, just have a look at the .NET Runtime repo on GitHub and search for the term SearchValue. You will see that Microsoft is using the new **`SearchValue`** type quite a bit.


repo:dotnet/runtime SearchValues
Filter by
Code 93
Issues 37
Pull requests 43
Discussions 1
Commits 20
Packages 0
Wikis 0
93 files (569 ms) in dotnet/runtime
src/libraries/System.Private.CoreLib/src/System/SearchValues/SearchValues.T.cs
Show 6 more matches

```
/// </remarks>
[DebuggerDisplay("{DebuggerDisplay,nq}")]
[DebuggerTypeProxy(typeof(SearchValuesDebugView<>))]
public class SearchValues<T> where T : IEquatable<T>?
{
    // Only CoreLib can create derived types
    private protected SearchValues() { }
```

Figure 20: The .NET Runtime Using SearchValues

This means that your code will also get this performance improvement for free, just by upgrading to .NET 8.

Extension libraries

.NET 8 has also given some love to extension libraries. You might have heard about keyed dependency injection (DI), where you use keys to register and read DI services. Secondly, the **`IHostedLifecycleService`** has been extended to include additional methods to give you more options for execution. The **`System.Numerics.Tensors`** NuGet package has also been updated to include the new **`TensorPrimitives`** namespace. This adds support for tensor operations. Tensor primitives optimize data-intensive workloads used in AI and machine learning.

Keyed DI services

As you know, the DI service is a way to register and retrieve services from the dependency injection container. Now, with keyed dependency injection, you can do this using keys. This allows you to register and inject multiple services of the same type using a unique key for each. Some of the new APIs include the following:

- The interface **`IKeyedServiceProvider`**.
- The attribute **`ServiceKeyAttribute`**, used to inject the key used for registration/resolution in the constructor.
- The attribute **`FromKeyedServicesAttribute`**, used to specify which keyed service to use in the service constructor parameter.
- New extension methods for **`IServiceCollection`** to support keyed services.
- The **`ServiceProvider`** implementation of **`IKeyedServiceProvider`**.

The easiest way to illustrate keyed dependency injection is through the use of minimal APIs. I admit that I haven't found myself gravitating towards using minimal APIs on a regular basis, but for this explanation, they're perfect.

Note: I have included the minimal API code illustrated here in the project KeyedDemoMinimal, but I have also added this code to an API project called KeyedDemo that uses controllers. You can find all this code in the GitHub repository that comes with this book.

The code in Figure 20 illustrates the use of keyed DI services. It might make a little more sense when viewed in a code editor, as the code is a little bit squished on the page. If you prefer using controllers instead, there is sample code for that, too.

The code contains an interface called **`IGreeter`** that is used to provide the contract for two services, **`FormalGreeterService`** and **`InformalGreeterService`**. I then have two other services, called **`CustomerService`** and **`FriendsService`**.

Adding the services to the DI container is done in the usual manner for the **`Customer`** and **`FriendsService`**, but for the **`FormalGreeterService`** and **`InformalGreeterService`**, I provide each with a key.

These keys allow me to reference each service uniquely and provide API endpoints that make use of the key to provide the appropriate service. You can see this in the **`api/GreetFriendSpecific`** and **`api/GreetCustomerSpecific`** endpoints.

Code Listing 20: The Keyed DI Minimal API

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
builder.Services.AddKeyedSingleton<IGreeter,
FormalGreeterService>("customers");
builder.Services.AddKeyedSingleton<IGreeter,
InformalGreeterService>("friends");
builder.Services.AddSingleton<CustomerService>();
builder.Services.AddSingleton<FriendsService>();

var app = builder.Build();

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

app.UseHttpsRedirection();


app.MapGet("api/GreetCustomer", (CustomerService customer) =>
customer.Greet());

app.MapGet("api/GreetFriend", (FriendsService friend) => friend.Greet());

app.MapGet("api/GreetFriendSpecific",
    ([FromKeyedServices("friends")] IGreeter greet, string friendName) =>
    greet.Greeting($"Hey {friendName}. Nice to see you."));

app.MapGet("api/GreetCustomerSpecific",
    (HttpContext httpContext, string customerName) =>
    httpContext.RequestServices
    .GetRequiredKeyedService<IGreeter>("customers")
    .Greeting($"Good day {customerName}. How many I assist you?"));

app.Run();

public class CustomerService([FromKeyedServices("customers")] IGreeter
greeter)
```

```
{
    public string Greet() => greeter.Greeting("Good day, can I be of 
service?");
}

public class FriendsService(IServiceProvider serviceProvider)
{
    public string Greet() => 
serviceProvider.GetRequiredKeyedService<IGreeter>("friends").Greeting("Hell
o buddy!");
}



public interface IGreeter
{
    string Greeting(string message);
}

public class FormalGreeterService : IGreeter
{
    public string Greeting(string message) => $"Formal greeting: 
{message}";
}

public class InformalGreeterService : IGreeter
{
    public string Greeting(string message) => $"Informal greeting: 
{message}";
}
```

But hold on a minute—didn't I say that keyed DI allows you to register and inject multiple services of the same type using a unique key for each? Well, yes, I did, and you can do this by adding the keyed services seen in Code Listing 21 to your DI container.

Code Listing 21: Adding Keyed Services of the Same Type

```
builder.Services.AddKeyedSingleton<IGreeter, FormalGreeterService>("ceo");
builder.Services.AddKeyedSingleton<IGreeter, FormalGreeterService>("cto");
```

I am giving the service a different key: one for CEO, and one for CTO. Now I can go ahead and add the following API endpoints, as illustrated in Code Listing 22.

Code Listing 22: API Endpoints

```
app.MapGet("api/GreetCEO",
    ([FromKeyedServices("ceo")] IGreeter greet) =>
    greet.Greeting($"Good day Mr CEO. How many I assist you?"));

app.MapGet("api/GreetCTO",
    ([FromKeyedServices("cto")] IGreeter greet) =>
    greet.Greeting($"Good day Mr CTO. How many I assist you?"));
```

I can now call the services by using their keys. While the implementation of the services does not differ, the idea is that you can register the exact same service with different keys and still have valid code.

Options validation

Validating your application settings is a very useful feature in .NET. In fact, being able to check that settings are within specific ranges, of a specific value or enum, allows for explicit control over the settings that your application needs to perform optimally. Before .NET 8, this options validation used reflection. This kind of reflection on application startup will generally cause your application to start slower.

With the release of .NET 8, Microsoft gave us a brand-new source generator that will generate code upfront to perform this validation.

Note: While the validation code is generated at compile time, it is worth noting that the actual validation still happens at runtime.

Therefore, instead of using reflection, it uses the generated validation code, which is optimized by Microsoft's source generators. You can be sure that they spent a lot of time creating generators that will generate the most optimized and high-performance code possible. Let's have a look at a concrete example of how this would work.

Looking at Code Listing 23, I have an appsettings.json file that has a section called **`ApiOptions`**. These are the settings that I want to validate.

Code Listing 23: The appsettings.json File

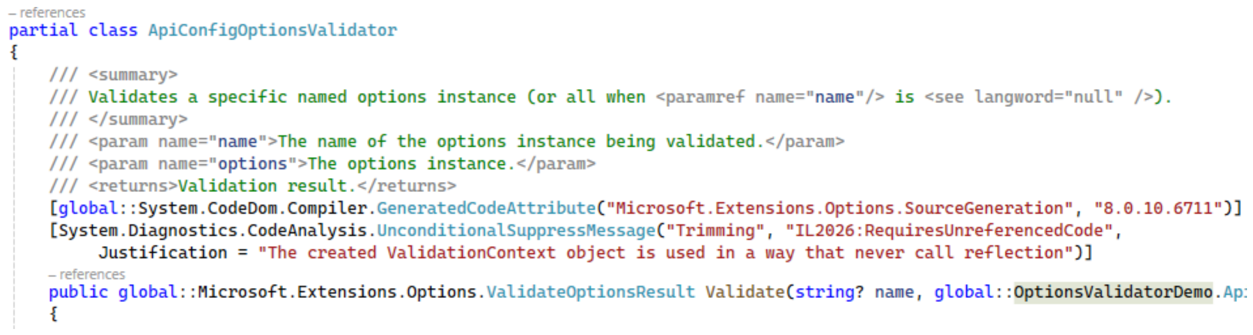
```
{
  "ApiOptions": {
    "NotificationType": "Email",
    "NotificationEmailAddress": "noreply@acme.com",
    "Attempts": 2
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

I then created a class called **ApiConfigOptions**, which you can see in Code Listing 24. This class contains properties for each setting. You will notice that at the end of the class, I have added another partial class called **ApiConfigOptionsValidation**, which implements **IValidateOptions<ApiConfigOptions>** without implementing a **Validate** method.

Code Listing 24: The Validation Class

```
using System.ComponentModel.DataAnnotations;
using Microsoft.Extensions.Options;

namespace OptionsValidatorDemo
{
    enum NotificationType
    {
        Email,
        Sms,
        Push
    }

    public class ApiConfigOptions
    {
        public const string SectionName = "ApiOptions";

        [EnumDataType(typeof(NotificationType)
            , ErrorMessage = "Invalid notification type defined.")]
        public required string NotificationType { get; init; }

        [Required]
        [RegularExpression(@"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-
Z]{2,}$"
            , ErrorMessage = "Invalid notification email address
defined")]
        public required string NotificationEmailAddress { get; init; }


        [Required]
        [Range(1, 3
            , ErrorMessage = "The value for Attempts is outside the valid 
range.")]
        public required int Attempts { get; init; }
    }

    [OptionsValidator]
    public partial class  ApiConfigOptionsValidator : 
IValidateOptions<ApiConfigOptions>
    {

    }
}
```

I didn't have to implement a **`Validate`** method because I decorated the partial class with an **`[OptionsValidator]`** attribute. Placing your mouse on the **`ApiConfigOptionsValidator`** class and hitting F12 will allow you to navigate to a file called Validators.g.cs, where you can see the generated code. It will not make much sense to illustrate the generated code here in a code block, but I want to highlight some areas of this generated code with some images from the file.



```
– references
partial class ApiConfigOptionsValidator
{
    /// <summary>
    /// Validates a specific named options instance (or all when <paramref name="name"/> is <see langword="null" />).
    /// </summary>
    /// <param name="name">The name of the options instance being validated.</param>
    /// <param name="options">The options instance.</param>
    /// <returns>Validation result.</returns>
    [global::System.CodeDom.Compiler.GeneratedCodeAttribute("Microsoft.Extensions.Options.SourceGeneration", "8.0.10.6711")]
    [System.Diagnostics.CodeAnalysis.UnconditionalSuppressMessage("Trimming", "IL2026:RequiresUnreferencedCode",
         Justification = "The created ValidationContext object is used in a way that never call reflection")]
    – references
    public global::Microsoft.Extensions.Options.ValidateOptionsResult Validate(string? name, global::OptionsValidatorDemo.Ap
    {
```

Figure 21: Generated Validation Code Comments

You can see from Figure 21 that Microsoft is explicitly stating that the **`ValidationContext`** is used in such a way as to never call reflection. Do yourself a favor and take a look at the generated code. It's fascinating to see what was generated, and how this changes as you add more properties to the **`ApiConfigOptions`** class.

Lastly, we want to validate the settings and call the API endpoint. In the Program.cs file, you can see that I have registered a configuration instance for the **`ApiConfigOptions`** class. I have also added the **`IValidateOptions`** service to the DI container as a singleton.

With this in place, I can add the code to access the options object and set it to a variable called **`settingOptions`**. It is at this point that your code will execute the generated code to perform options validation.

Code Listing 25: Validating and Calling the API

```
using Microsoft.Extensions.Options;
using OptionsValidatorDemo;

var builder = WebApplication.CreateBuilder(args);
var cfg = builder.Configuration;
// Add services to the container.

builder.Services
    .Configure<ApiConfigOptions>(cfg.GetSection(ApiConfigOptions.SectionNam
e));

builder.Services
    .AddSingleton<IValidateOptions<ApiConfigOptions>,
ApiConfigOptionsValidator>();

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();

var settingOptions =
app.Services.GetRequiredService<IOptions<ApiConfigOptions>>().Value;


app.MapGet("api/sendnotification", () =>
{
    var notificationType = settingOptions.NotificationType;
    return Enum.TryParse<NotificationType>(notificationType, out var type)
        ? type switch
        {
            NotificationType.Email => "Sending email notification",
            NotificationType.Sms => "Sending sms notification",
            NotificationType.Push => "Sending push notification",
            _ => "Invalid notification type",
        }
        : "Notification not sent due to undefined notification type";
})
.WithName("SendNotification")
.WithDescription("Send notification to the user")
.WithOpenApi();

app.Run();
```

If any of the setting values in the appsettings.json file are incorrect according to the constraints set in the validator class, you will see an exception. Compare Code Listing 23 to Code Listing 26. You will notice that I have modified the value for **`Attempts`** to a value beyond the allowed range set on the **`Attempts`** property in the **`ApiConfigOptions`** class.

Code Listing 26: The Modified Settings

```
{
  "ApiOptions": {
    "NotificationType": "Email",
    "NotificationEmailAddress": "noreply@acme.com",
    "Attempts": 20
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

Run your API and have a look at the result. The validation kicks in, and the error message informs you that the value specified for this property is not valid.

If, however, you opt to inject your **`IOptions`** object on your API endpoints, you will not get an error when the application starts, but rather on the API call. A good suggestion would be to create a health check endpoint to validate your settings.

Code Listing 27 illustrates how you might achieve this. A way to validate your settings is, more often than not, a good idea. This is especially true if you have production code that needs a quick and easy way to check that the basics are still correct after a deployment. Sometimes the fog of war tends to allow mistakes to slip in, and a way to validate that your API is still healthy quickly is to include some sort of check.

Code Listing 27: Adding a Settings Validation Endpoint

```
using System.Text.Json;
using Microsoft.Extensions.Options;
using OptionsValidatorDemo;

var builder = WebApplication.CreateBuilder(args);
var cfg = builder.Configuration;
// Add services to the container.

builder.Services
    .Configure<ApiConfigOptions>(cfg.GetSection(ApiConfigOptions.SectionNam
e));

builder.Services
    .AddSingleton<IValidateOptions<ApiConfigOptions>,
ApiConfigOptionsValidator>();

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.MapGet("api/sendnotification", (IOptions<ApiConfigOptions> opt) =>
{
    var notificationType = opt.Value.NotificationType;
    return Enum.TryParse<NotificationType>(notificationType, out var type)
        ? type switch
        {
            NotificationType.Email => "Sending email notification",
            NotificationType.Sms => "Sending sms notification",
            NotificationType.Push => "Sending push notification",
            _ => "Invalid notification type",
        }
        : "Notification not sent due to undefined notification type";
})
.WithName("SendNotification")
.WithDescription("Send notification to the user")
.WithOpenApi();
```

```
// Validate settings.
app.MapGet("api/validatesettings", (IOptions<ApiConfigOptions> opt) =>
{
    var options = new JsonSerializerOptions { WriteIndented = true };

    try
    {
        return JsonSerializer.Serialize(new { status = "OK", settings =
opt.Value }, options);
    }
    catch (Exception ex)
    {
        return JsonSerializer.Serialize(new { status = "ERROR", error =
ex.Message }, options);
    }
})
.WithName("ValidateSettings")
.WithDescription("Perform a Check On The App Settings")
.WithOpenApi();

app.Run();
```

I have digressed a bit on the options validation feature in .NET 8. However you decide to implement options validation, .NET 8 allows your code to do this in a more flexible, manageable, and high-performance manner than the previous reflected code in previous .NET versions.

LoggerMessageAttribute constructors

When it comes to logging, the **`LoggerMessageAttribute`** offers additional overloads on its constructor. In previous versions of .NET, you had to take an all-or-nothing approach. Either you had to use the parameterless constructor, or you had to choose to supply all the required parameters for event ID, log level, and message.

Code Listing 28: LoggerMessageAttribute Constructor Overloads

```
public LoggerMessageAttribute(LogLevel level, string message);
public LoggerMessageAttribute(LogLevel level);
public LoggerMessageAttribute(string message);
```

As seen in Code Listing 28, developers have more options when specifying the required parameters due to these overloads. If you don't, for example, specify an event ID, one will be automatically generated.

C# 12

Most excitingly of all, C# 12 shipped with .NET 8. The following new features were included in C#12:

- Primary constructors
- Collection expressions
- Ref readonly parameters
- Default lambda parameters
- Alias any type
- Inline arrays
- Experimental attribute
- Interceptors

We will have a closer look at these in the next chapter.

.NET Aspire

You might have heard about .NET Aspire if you watched any of the dotnet conf or Build content online. .NET Aspire is available by including a collection of NuGet packages that handle specific cloud-native concerns. Currently in preview, it is available with .NET 8. Microsoft describes .NET Aspire as an “opinionated” (which makes no sense to me, so I prefer “highly directive”), cloud-ready stack for building observable, production-ready, distributed applications.

Simply put, if you are creating distributed applications, .NET Aspire will solve a lot of your pain points. The main problem with many small distributed applications is that we want to have them all talk to each other, both locally in the dev environment and when we deploy them. This is what .NET Aspire aims to solve.

This book is too short to go into an in-depth explanation of .NET Aspire, which deserves a book on its own. If you want to learn more about .NET Aspire, head over to the .NET Aspire documentation.

In conclusion

This chapter introduced us to some of the goodies in .NET 8, and there is a lot that I have left out. There is ASP.NET Core with improvements to Blazor, SignalR, minimal APIs, Native AOT, Kestrel, authentication, and authorization. There is .NET MAUI, which includes new functionality for controls as well as performance enhancements. EF Core made improvements to complex type objects, collections of primitive types, raw SQL queries, tracked-entity access, and much more. Windows Forms got some love, too, as well as Windows Presentation Foundation, which now has the ability to use hardware acceleration and also has a new `OpenFolderDialog` control.

Buckle up and hold on to your seat, as this train is just gaining traction. Next up, we will have a look at C# 12 and all the new features introduced.

Chapter 2 A Closer Look at C# 12

C# 12 has brought with it a host of new features. With the pace at which C# is progressing, it remains a challenge for developers to stay up to date. The new features of C# 12 include:

- Primary constructors
- Collection expressions
- Ref readonly parameters
- Default Lambda parameters
- Alias any type
- Inline arrays
- Experimental attribute
- Interceptors

While developers might feel the pressure of staying up to date with everything that is new with every release of .NET and C#, I have come to a different conclusion. Instead of seeing it as a five-course meal that needs to be worked through in a single sitting, I am approaching all these new features in each new release like a buffet table.

A smorgasbord of C# and .NET features presented as a grand spread, meticulously prepared by Microsoft to satisfy the eclectic tastes of the most discerning code connoisseurs.

“Yes, thank you very much,” I murmur as I slowly pace along the buffet with my dinner plate, perusing all that is on display.

As I saunter along the buffet table (taking care not to walk too fast), my plate ready for the taking, I whisper to myself: “A little bit of record structs, a dash of global using directives, a slice of file-scoped namespaces, topped with some file-scoped types… Oohh, primary constructors and collection expressions, haven’t tried you before!”

I carefully scoop just enough onto my plate of each so that I don’t look greedy. You see, the trick to navigating any buffet table is to put something on your plate, but also to pop something surreptitiously into your mouth (from your plate, not directly from the table) while standing at the table. This way, you get to taste more while only returning to your table with a modest plate of food. The same can be said for the flurry of new features with every release of .NET.

You don’t have to learn to implement every new feature in every release of .NET in every project you create. What you do have to do, however, is peruse the buffet table. You must walk up to it, pick something up here, nibble something there, and return to your table with a manageable plate of food. While this metaphor does well to illustrate how I can grow my skills as a developer, and when life imitates art, grow my waistline as a buffet table peruser, it does not account for the challenges developers face in the real world. Trying to grow as a developer while managing a healthy work-life balance is challenging when you try to do it all at once. But it starts with showing up and taking little (pun intended), bite-sized nibbles.

As Carl Orff’s “O Fortuna” from Carmina Burana starts playing in our minds, the waiters, ready with the champagne, join me as we walk up to the buffet table to see what’s on display.

Primary constructors

I'm not convinced I like primary constructors. My mind isn't yet made up. In fact, it feels like it goes against my flow of writing code. Let me show you what I mean.

Code Listing 29 illustrates a weather service that generates totally random weather conditions for random locations every time you call it.

Code Listing 29: The LyingWeatherController

```
using Microsoft.AspNetCore.Mvc;
using PrimaryConstructorsDemo.Services;

namespace PrimaryConstructorsDemo.Controllers;

[ApiController]
[Route("[controller]")]
public class LyingWeatherForecastController : ControllerBase
{
    private readonly IRandomCityService _randomCityService;
    private readonly IRandomSummaryService _randomSummaryService;

    public LyingWeatherForecastController(IRandomCityService
randomCityService, IRandomSummaryService randomSummaryService)
    {
        _randomCityService = randomCityService;
        _randomSummaryService = randomSummaryService;
    }

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

It happens to be correct about 1 percent of the time. Nevertheless, I like this method of creating classes. I have a constructor, I inject some services, I set those to private fields, and I use the services in my class. End of story.

Let’s take a look at the same class using primary constructors. The parameters on the constructor are moved up to the class level. Now I can remove my constructor totally, as well as the private fields, and use the injected services in my code.

Code Listing 30: Using Primary Constructors

```
using Microsoft.AspNetCore.Mvc;
using PrimaryConstructorsDemo.Services;

namespace PrimaryConstructorsDemo.Controllers;

[ApiController]
[Route("[controller]")]
public class LyingWeatherForecastController(
    IRandomCityService randomCityService,
    IRandomSummaryService randomSummaryService) : ControllerBase
{

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

This resulting code takes some getting used to, that’s for sure.

What the compiler sees

Let’s simplify the code and see what the lowered C# looks like. Code Listing 31 illustrates a simplified **`LyingWeatherForecastController`** class.

Code Listing 31: The Simplified Class

```
public class LyingWeatherForecastController(
        IRandomCityService randomCityService,
        IRandomSummaryService randomSummaryService) : IWidget
{
        public string GetSomeWeatherData()
        {
            var city = randomCityService.GetRandomCity();
            var summary = randomSummaryService.GetRandomSummary();
            return $"It is {summary} in {city}";
        }
}
```

Have a look what the lowered C# code looks like in Code Listing 32. Notice that the private fields for **`IRandomCityService`** and **`IRandomSummaryService`** are not **`readonly`**.

Code Listing 32: The Lowered C# Code

```
public class LyingWeatherForecastController : IWidget
{
    [CompilerGenerated]
    [DebuggerBrowsable(DebuggerBrowsableState.Never)]
    private IRandomCityService <randomCityService>P;

    [CompilerGenerated]
    [DebuggerBrowsable(DebuggerBrowsableState.Never)]
    private IRandomSummaryService <randomSummaryService>P;

    public LyingWeatherForecastController(IRandomCityService
randomCityService, IRandomSummaryService randomSummaryService)
    {
        <randomCityService>P = randomCityService;
        <randomSummaryService>P = randomSummaryService;
        base..ctor();
    }

    public string GetSomeWeatherData()
    {
        string randomCity = <randomCityService>P.GetRandomCity();
        string randomSummary =
<randomSummaryService>P.GetRandomSummary();
        return string.Concat("It is ", randomSummary, " in ",
randomCity);
    }
}
```

It is, therefore, possible to change the value of these fields from inside the class—so watch out for that little pitfall.

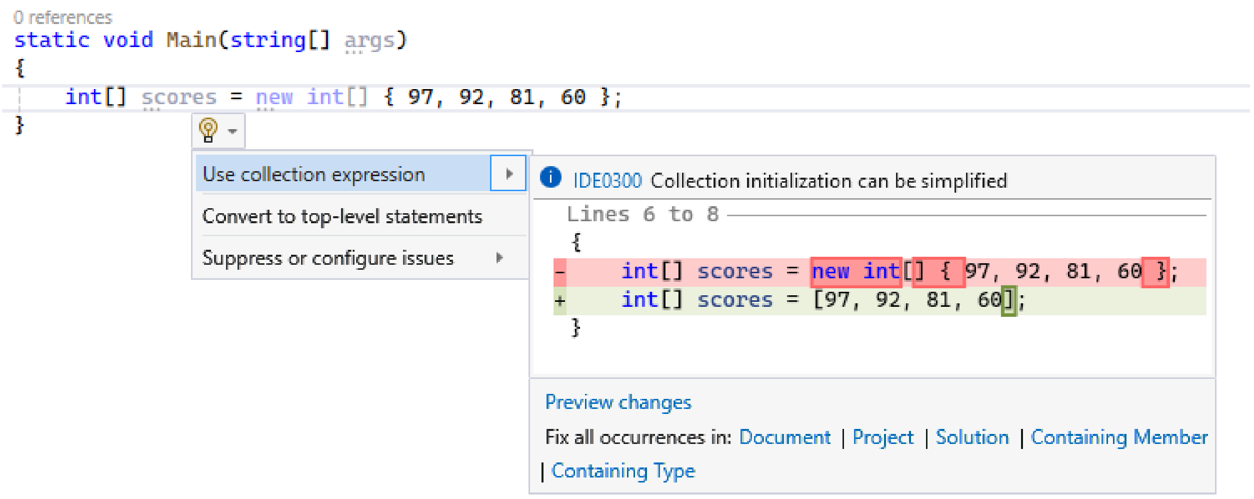
Collection expressions

I have to say that I do like collection expressions. In my opinion, it weeds out a little bit of the unnecessary fluff. Of course, the argument can be made that it's not as expressive, but here we are. Let's look at an example.

Code Listing 33: A Simple Array

```
int[] scores = new int[] { 97, 92, 81, 60 };
```

As seen in Code Listing 33, we have a stock standard array of integers. Visual Studio, however, will suggest a quick action to use collection expressions, as seen in Figure 22.



```
0 references
static void Main(string[] args)
{
    int[] scores = new int[] { 97, 92, 81, 60 };
}
```

Use collection expression
Convert to top-level statements
Suppress or configure issues

IDE0300 Collection initialization can be simplified

```
Lines 6 to 8
{
-    int[] scores = new int[] { 97, 92, 81, 60 };
+    int[] scores = [97, 92, 81, 60];
}
```

Preview changes
Fix all occurrences in: Document | Project | Solution | Containing Member | Containing Type

Figure 22: Refactoring Quick Action

You can see the resulting code in Code Listing 34.

Code Listing 34: The Collection Expression

```
int[] scores = [97, 92, 81, 60];
```

Just also note that you can't use **`var`** with collection expressions, as then there is no target type. Collection expressions also apply to **`Span<T>`**, as seen in Code Listing 35.

Code Listing 35: Span Collection Expression

```
Span<int> foo = ['a', 'b', 'c'];
```

Collection expressions may appeal to some developers but not to others. With C# 12, you have the option to use collection expressions if you choose to.

Ref readonly parameters

Ref readonly parameters are now a thing in C# 12. Consider the code in Code Listing 36. This simply passes a **`score`** to the **`Increment`** method of the **`Counter`** class and then prints out the score.

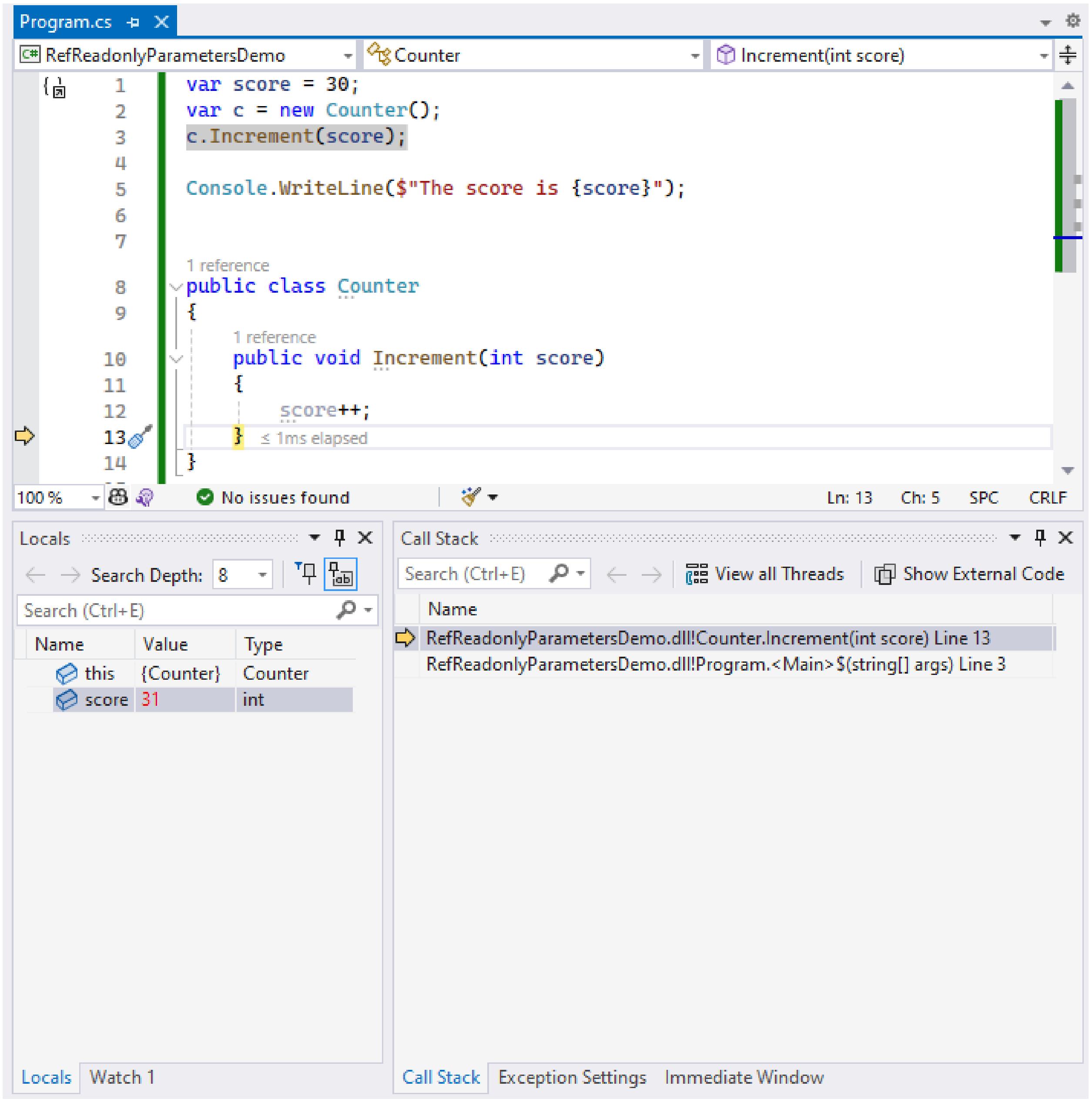
Code Listing 36: Arbitrary Code Example

```
var score = 30;
var c = new Counter();
c.Increment(score);

Console.WriteLine($"The score is {score}");


public class Counter
{
    public void Increment(int score)
    {
        score++;
    }
}
```

As you would expect, the value of **`score`** remains unchanged. Why is this the case? Well, the **`score`** parameter is passed by value. In other words, a copy of the variable is made, and the method acts on this copy. Therefore, changes to this copy inside the method will not affect the original variable.

Tip: Value types are copied by default.

We can see this behavior when we inspect the call stack, as seen in Figure 23.



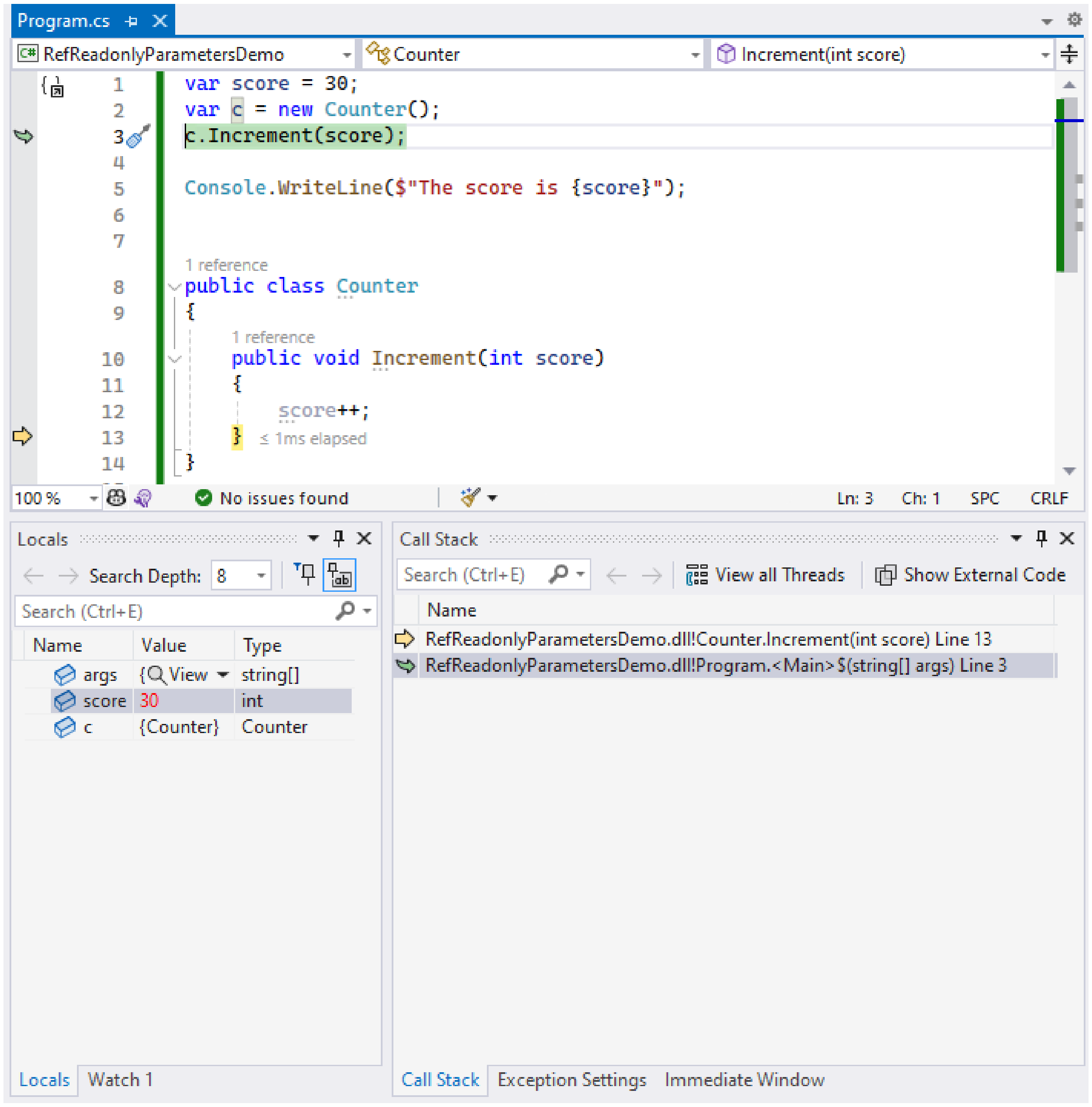
```
var score = 30;
var c = new Counter();
c.Increment(score);

Console.WriteLine($"The score is {score}");


public class Counter
{
    public void Increment(int score)
    {
        score++;
    }
}
```

Name	Value	Type
this	{Counter}	Counter
score	31	int

Name
RefReadonlyParametersDemo.dll!Counter.Increment(int score) Line 13
RefReadonlyParametersDemo.dll!Program.<Main>$(string[] args) Line 3

Figure 23: Value of score Variable Inside Increment

When the code execution reaches the **`Increment`** method and increments the **`score`** parameter, the value changes to **`31`**. As seen in Figure 24, a whole different story is at play in the calling code.



```
var score = 30;
var c = new Counter();
c.Increment(score);

Console.WriteLine($"The score is {score}");


1 reference
public class Counter
{
    1 reference
    public void Increment(int score)
    {
        score++;
    } ≤ 1ms elapsed
}
```

Name	Value	Type
args	{View	string[]
score	30	int
c	{Counter}	Counter

Name
RefReadonlyParametersDemo.dll!Counter.Increment(int score) Line 13
RefReadonlyParametersDemo.dll!Program.<Main>$(string[] args) Line 3

Figure 24: Value of score Variable in Main

The **`score`** variable's value in the calling code remains unchanged at **`30`**. This is because, as mentioned previously, the increment acted on a copy of this variable.

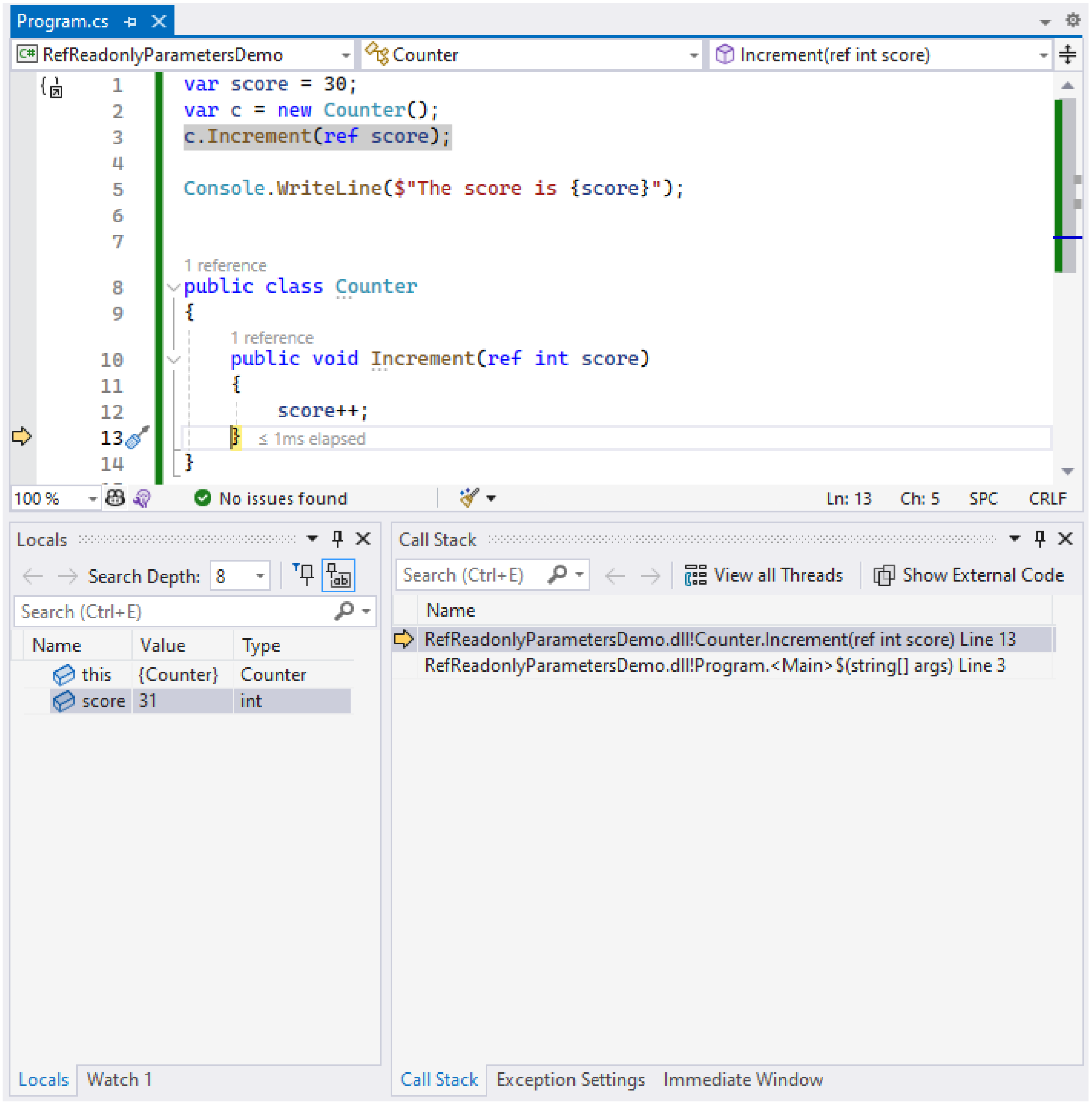
In order to change this behavior, we can add the **`ref`** keyword to the parameter and the calling code, as seen in Code Listing 37. This allows me to pass down the reference to the score variable instead of the value of the score variable.

Code Listing 37: Arbitrary Code Example Using Ref readonly Parameters

```
var score = 30;
var c = new Counter();
c.Increment(ref score);

Console.WriteLine($"The score is {score}");


public class Counter
{
    public void Increment(ref int score)
    {
        score++;
    }
}
```

What does this do to our variable, you might wonder.

As soon as the variable is incremented, I am increasing the reference of the variable **`score`**, not the value itself. This means that we will see this effect outside the **`Increment`** method in the calling code.



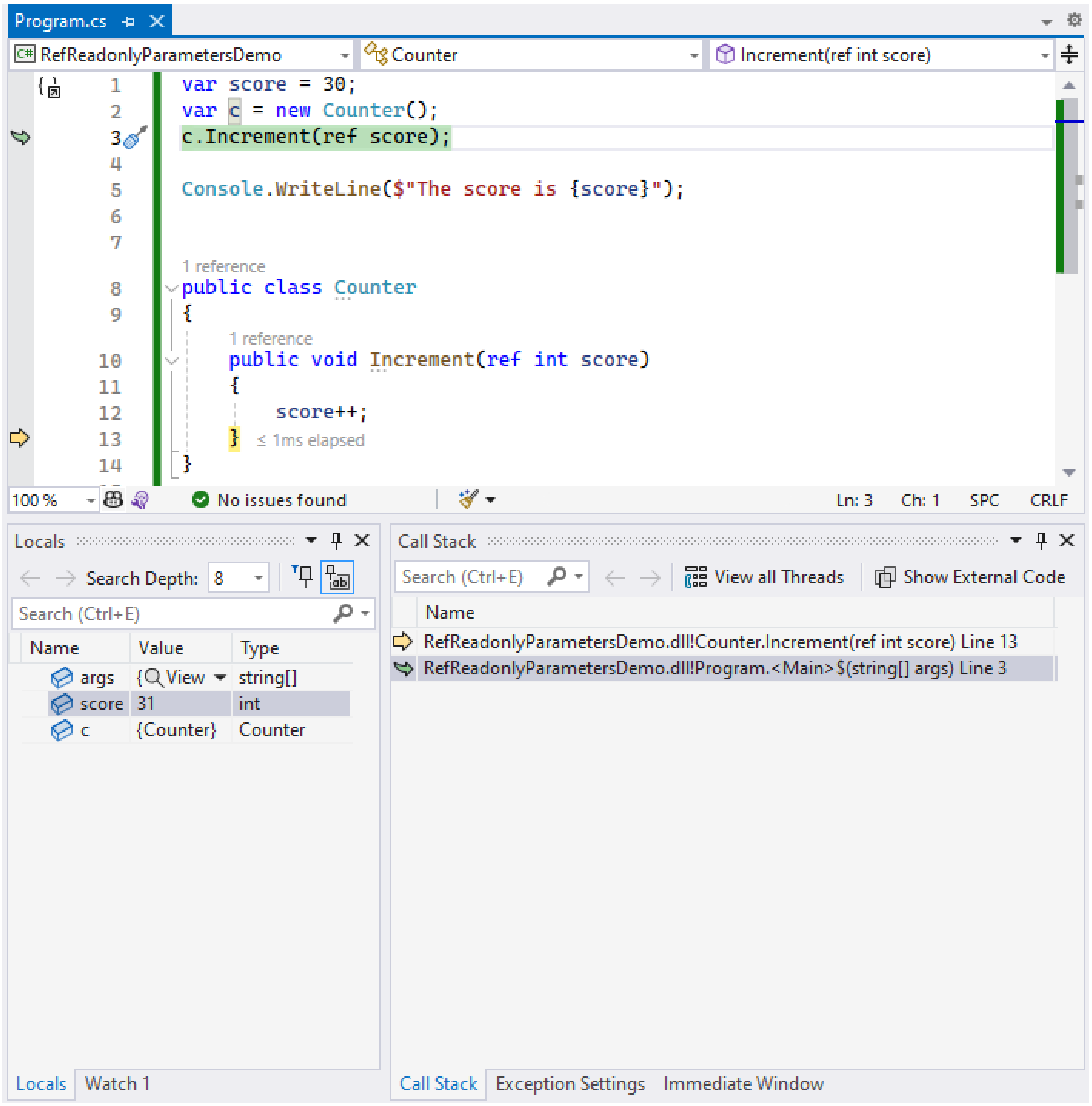
```
var score = 30;
var c = new Counter();
c.Increment(ref score);

Console.WriteLine($"The score is {score}");


1 reference
public class Counter
{
    1 reference
    public void Increment(ref int score)
    {
        score++;
    } ≤ 1ms elapsed
}
```

Name	Value	Type
this	{Counter}	Counter
score	31	int

Name
RefReadonlyParametersDemo.dll!Counter.Increment(ref int score) Line 13
RefReadonlyParametersDemo.dll!Program.<Main>$(string[] args) Line 3

Figure 25: Value of score Variable Inside Increment

Figure 25 illustrates this point when looking at the call stack again. As soon as the `score` parameter is incremented, the value increases to **`31`**, as expected. The difference now, because we used the **`ref`** keyword, is that the original variable in the calling code has also increased, as seen in Figure 26.



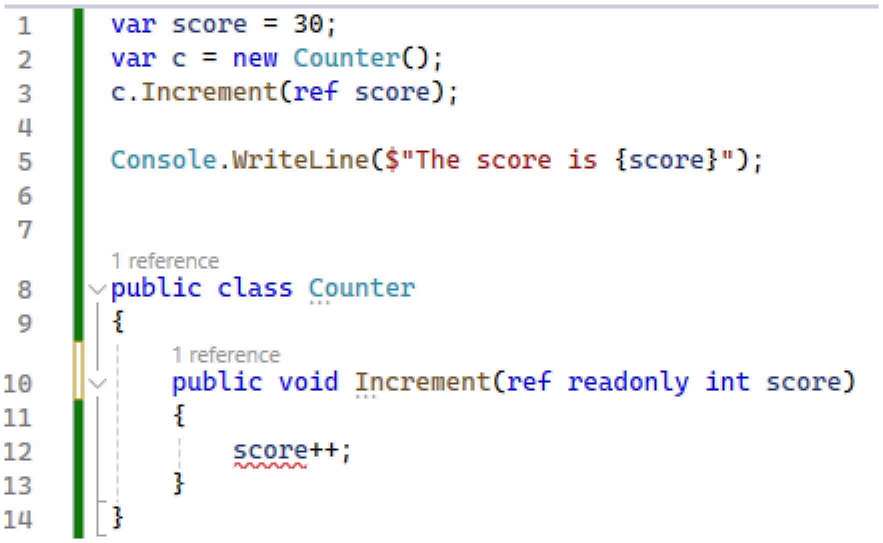
```
var score = 30;
var c = new Counter();
c.Increment(ref score);

Console.WriteLine($"The score is {score}");


1 reference
public class Counter
{
    1 reference
    public void Increment(ref int score)
    {
        score++;
    } ≤ 1ms elapsed
}
```

Name	Value	Type
args	{View	string[]
score	31	int
c	{Counter}	Counter

Name
RefReadonlyParametersDemo.dll!Counter.Increment(ref int score) Line 13
RefReadonlyParametersDemo.dll!Program.<Main>$(string[] args) Line 3

Figure 26: Value of score Variable in Main

With C# 12, however, you can add the **`readonly`** keyword, as seen in Figure 27.



```
var score = 30;
var c = new Counter();
c.Increment(ref score);

Console.WriteLine($"The score is {score}");


1 reference
public class Counter
{
    1 reference
    public void Increment(ref readonly int score)
    {
        score++;
    }
}
```

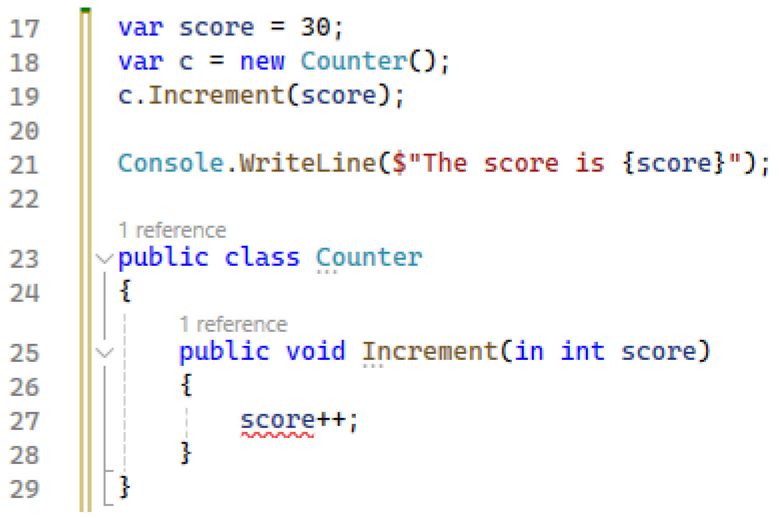
Figure 27: Using Ref readonly

Using the **`ref readonly`** keywords in the **`Increment`** method will result in a compilation error telling you that the reference passed is read-only, and that you can't change it. Now you might be wondering, why not just use the **`in`** keyword, and why do we have **`ref readonly`** to begin with?

Code Listing 38: Using the in Keyword

```
public void Increment(in int score)
{
    score++;
}
```

Code Listing 38 illustrates the use of the **`in`** keyword. Looking at Figure 28, you will see that this also results in a compilation error.



```
var score = 30;
var c = new Counter();
c.Increment(score);

Console.WriteLine($"The score is {score}");

1 reference
public class Counter
{
    1 reference
    public void Increment(in int score)
    {
        score++;
    }
}
```

Figure 28: The Result Is the Same Compilation Error

So then, why add another feature to C# 12 that does the same thing introduced in C# 7.2 with the **`in`** keyword?

Note: You can read about the release of C# 7.2 here.

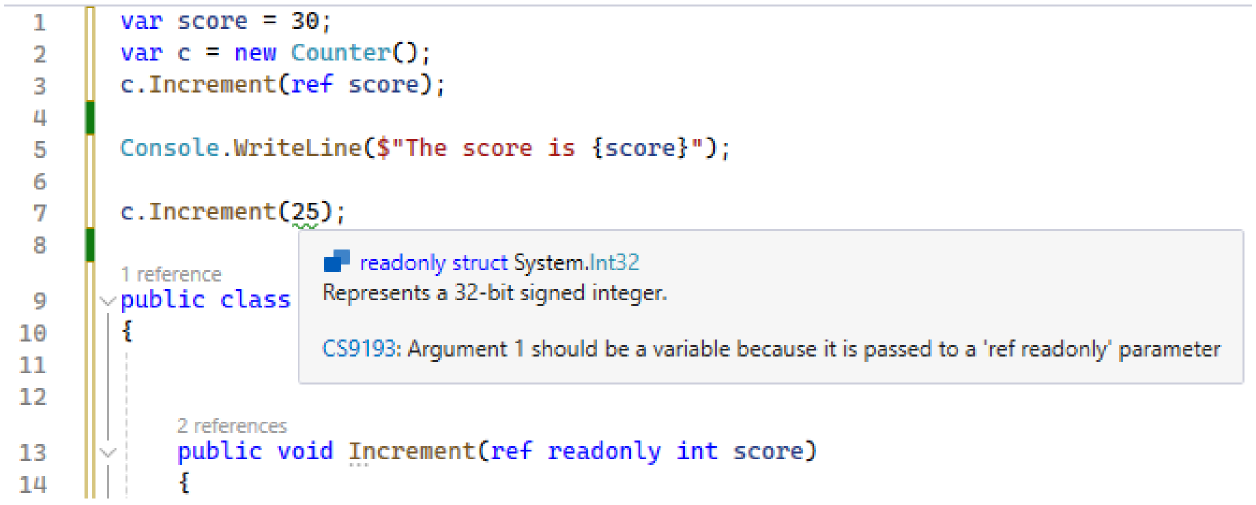
To explain the reason, we first need to understand the difference between an rvalue and an lvalue.

Rvalue stands for right-hand side value and refers to the value that appears on the right side of an assignment expression. In other words, it is a value that does not have a persistent memory location.

Lvalue, on the other hand, stands for left-hand side and appears on the left-hand side of an assignment expression. In other words, an lvalue typically represents a variable or object that can be assigned a new value.

Looking at the expression **`int x = 27;`** and applying the definitions for an lvalue and an rvalue, **`27`** is an rvalue, while **`x`** is an lvalue.

To put it simply, using **`ref readonly`** offers additional guarantees by warning that an rvalue (in other words, not a variable) is being passed. This enhances clarity and intent at the call site that a reference is being captured.


```
var score = 30;
var c = new Counter();
c.Increment(ref score);

Console.WriteLine($"The score is {score}");

c.Increment(25);

1 reference
public class
{


    2 references
    public void Increment(ref readonly int score)
    {
```

readonly struct System.Int32
Represents a 32-bit signed integer.

CS9193: Argument 1 should be a variable because it is passed to a 'ref readonly' parameter

Figure 29: Warning on rvalue

By introducing the **`ref readonly`** feature in C# 12, you can enforce this indication that a reference is being captured, and that the argument passed is a temporary value instead of a variable that exists beyond the method call (as seen in Figure 29).

Default lambda parameters

Before default lambda parameters, we had to invoke the lambda by specifying the value for **`personToGreet`**, as seen in Code Listing 39.

Code Listing 39: Before Default Lambda Parameters

```
var greeting = (string personToGreet) => $"Hello, {personToGreet}!";

Console.WriteLine(greeting("John")); // Hello, John!
```

In C# 12, however, we can now set a default value for **`personToGreet`**, as seen in Code Listing 40.

Code Listing 40: Using Default Lambda Parameters

```
var greeting = (string personToGreet = "World") => $"Hello,
{personToGreet}!";

Console.WriteLine(greeting()); // Hello, World!
```

This is a small, but welcome change.

Alias any type

In a nutshell, this feature is relaxing the rules on where the **`using`** alias directive can be used. In other words, you can now create semantic aliases for tuple types, array types, and pointer types where in the past, you could not. Previously, you could only alias named types, as seen in Code Listing 41.

Code Listing 41: Using Alias on Named Types

```
using Employee = System.Collections.Generic.Dictionary<string, string>;
using Foo = System.Console;

Employee employee = new()
{
    { "Name", "John Doe" }
};

Foo.WriteLine(employee["Name"]);
Foo.ReadLine();
```

With C# 12, however, you can now create an alias for a type that isn't a named type. You can create a **`using`** alias for a tuple, as seen in Code Listing 42.

Code Listing 42: Using Alias on a Tuple

```
using TuplePoints = (int x, int y);
using Foo = System.Console;

TuplePoints p = (3, 4);
Foo.WriteLine(p.x);
Foo.WriteLine(p.y);
Foo.ReadLine();
```

Being able to alias any type is useful because it reduces the amount of code you need to write, making your code more readable.

Experimental attribute

Library authors will most likely make use of the **`Experimental`** attribute the most. I'm not so sure that this can be classified as a feature in C# 12, but it's here, so let's have a look at it.

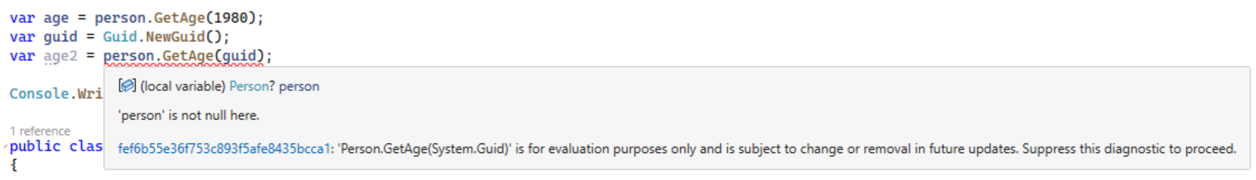
Code Listing 43: Using the Experimental Attribute

```
using System.Diagnostics.CodeAnalysis;
```

```
var person = new Person { Name = "John" };

var age = person.GetAge(1980);
var guid = Guid.NewGuid();
var age2 = person.GetAge(guid);

Console.WriteLine(age);

public class Person
{
    public string Name { get; set; }


    public int GetAge(int yearBorn)
    {
        // Do some standard calculation.
        // Just return default for now.

        return default;
    }

    [Experimental("fef6b55e36f753c893f5afe8435bcca1"
        , UrlFormat = "https://gist.github.com/dirkstrauss/{0}")]
    public int GetAge(Guid guid)
    {
        // Do advanced experimental calculation.
        // Just return default for now.

        return default;
    }
}
```

Code Listing 43 shows some boilerplate code for a class called **`Person`** that has two **`GetAge`** methods. The first method is one that is safe to use, but I have included another experimental method that is slightly risky to use.

The **`Experimental`** attribute allows me to decorate my second **`GetAge`** method and give it a diagnostic ID that the compiler can use to report any use of my method. It also allows me to set a URL for the corresponding documentation.



```
var age = person.GetAge(1980);
var guid = Guid.NewGuid();
var age2 = person.GetAge(guid);

Console.Wri

1 reference
public clas
{
```

(local variable) Person? person

'person' is not null here.

fef6b55e36f753c893f5afe8435bcca1: 'Person.GetAge(System.Guid)' is for evaluation purposes only and is subject to change or removal in future updates. Suppress this diagnostic to proceed.

Figure 30: The Error Message Displayed in Visual Studio

As seen in Figure 30, if I try to use this experimental feature, Visual Studio will display this warning to me, allowing me to click on the diagnostic ID that will take me to the relevant documentation for this warning. I have used a GUID, but you might be used to seeing these as compiler warnings in Visual Studio starting with "CS."

Note: You can take a look at some examples of these compiler warnings here.

In order for me to use this experimental feature, I need to explicitly suppress the warning in my code, as seen in Code Listing 44.

Code Listing 44: Suppressing the Warning

```
var guid = Guid.NewGuid();
#pragma warning disable fef6b55e36f753c893f5afe8435bcca1 // Type is for
evaluation purposes only and is subject to change or removal in future
updates. Suppress this diagnostic to proceed.
var age2 = person.GetAge(guid);
#pragma warning restore fef6b55e36f753c893f5afe8435bcca1 // Type is for
evaluation purposes only and is subject to change or removal in future
updates. Suppress this diagnostic to proceed.
```

I can also use the **`<NoWarn>`** csproj property to suppress this warning, as seen in Code Listing 45.

Code Listing 45: Using the <NoWarn> csproj Property

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <NoWarn>fef6b55e36f753c893f5afe8435bcca1</NoWarn>
  </PropertyGroup>

</Project>
```

Personally, I dislike the **#pragma** warning directive in code, but it provides a lot of visibility and expressiveness in the fact that you are suppressing a compiler warning. I would rather use the **#pragma** warning directive (even though I dislike it) as opposed to the **<NoWarn>** property in the csproj file. I feel that it's almost too hidden away in the csproj file.

There is an Afrikaans saying "*Stille waters, diepe grond, onder draai die duiwel rond*," which translates to "Still waters run deep, and that is where the devil lurks."

The devil in this detail is the **<NoWarn>** property in the csproj file suppressing a warning in the code editor that you spend most of your time in. If I have suppressed a warning, I want to know about it—and more importantly, I want other developers who contribute to my code to know about it, too.

Interceptors

One of the most exciting and fun features in C# 12 has got to be interceptors. Microsoft warns that interceptors are experimental and only available in preview mode with C# 12, and that this feature may be subject to breaking changes or removal in a future release.

It goes without saying that you are not encouraged to use this in production. There, I've said it. With that out of the way, let's give it a whirl.

You will see in Code Listing 46 that I have a class called **ArraySorter** with a horribly inefficient **Sort** method.

Tip: Do not use this* Sort *method in your code… ever.

This is basically a bubble sort, but it is not high-performance when having to sort large arrays.

Code Listing 46: The Old ArraySorter Class

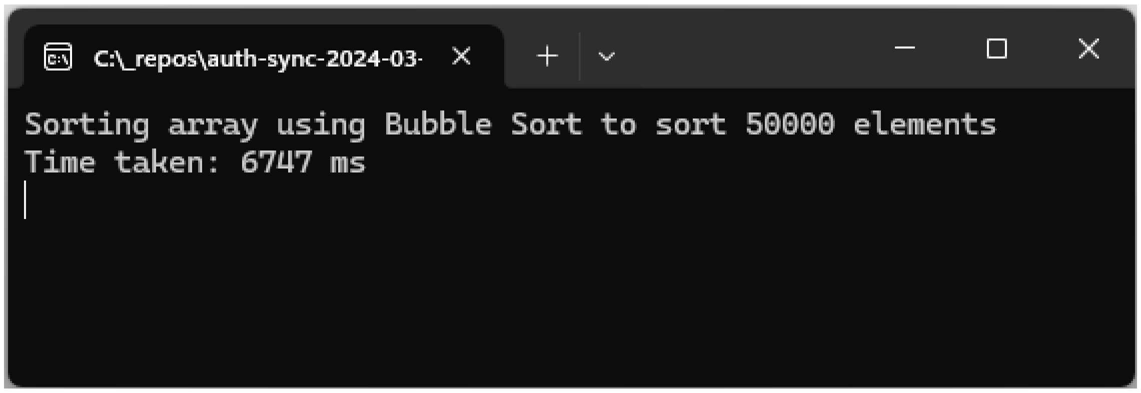
```
namespace InterceptorDemo;

public class ArraySorter
{
    public void Sort(int[] array)
    {
        Console.WriteLine($"Sorting array using Bubble Sort to sort 
{array.Length} elements");

        bool swapped;
        do
        {
            swapped = false;
            for (int i = 0; i < array.Length - 1; i++)
            {
                if (array[i] > array[i + 1])
                {
                    int temp = array[i + 1];
                    array[i + 1] = array[i];
                    array[i] = temp;
                    swapped = true;
                }
            }
        } while (swapped);
    }
}
```

In my Program.cs file, seen in Code Listing 47, I am creating an array of random integers.

Code Listing 47: The Program.cs Class

```
using System.Diagnostics;
using InterceptorDemo;

var random = new Random();
var largeArray = Enumerable.Range(0, 50000).Select(x =>
random.Next()).ToArray();

var sorter = new ArraySorter();

var stopwatch = Stopwatch.StartNew();

sorter.Sort(largeArray);

stopwatch.Stop();

Console.WriteLine($"Time taken: {stopwatch.ElapsedMilliseconds} ms");
Console.ReadLine();
```

I then instantiate my **`ArraySorter`** class and call the **`Sort`** method, passing it the **`largeArray`** variable.



```
Sorting array using Bubble Sort to sort 50000 elements
Time taken: 6747 ms
```

Figure 31: The Bubble Sort Results

The results of the current **`Sort`** method are displayed in Figure 31.

Tip: Do not increase the size of the `LargeArray` beyond 50,000 and run the bubble sort—you'll be here forever and a day.

What I want to do is hijack this inefficient **`Sort`** method. I want to implement a *coup d'état* and overthrow the bubble sort. And I want to do this without changing the calling code—otherwise, I couldn't call this an "interceptor" at all.

Code Listing 48: The csproj File

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <InterceptorsPreviewNamespaces>$(InterceptorsPreviewNamespaces);Interce
ptorDemo</InterceptorsPreviewNamespaces>
  </PropertyGroup>

</Project>
```

To start off, I need to modify the csproj file and opt in to use the experimental interceptors feature. Let's add the property **`<InterceptorsPreviewNamespaces>`** and include our **`InterceptorDemo`** namespace in here.

Next, we need to add the **`InterceptsLocationAttribute`**, which is crucial for implementing interceptors, as this is not yet available in our build of C# 12. You can see this in Code Listing 49. Put simply, this allows us to use the interceptor feature.

Code Listing 49: The InterceptsLocationAttribute Helper Class

```
namespace System.Runtime.CompilerServices
{
    [AttributeUsage(AttributeTargets.Method, AllowMultiple = false)]
    internal sealed class InterceptsLocationAttribute : Attribute
    {
        public InterceptsLocationAttribute(string filePath, int line, int
column) { }
    }
}
```

I then create a class called **`Inception`**, as seen in Code Listing 50.

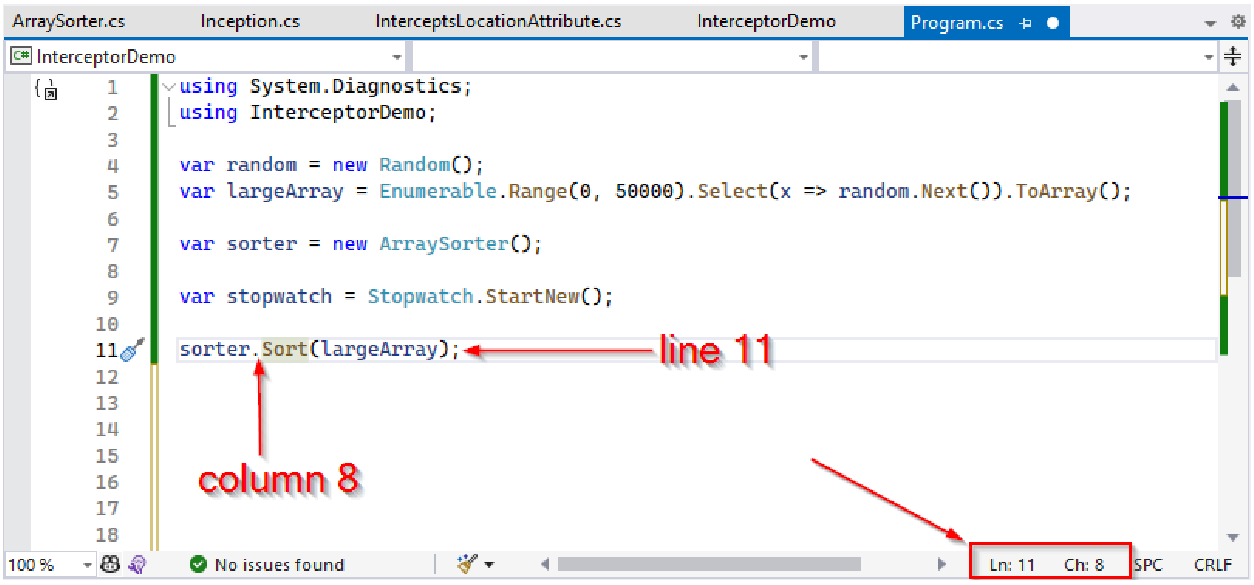
Code Listing 50: Our Interceptor

```
using System.Runtime.CompilerServices;

namespace InterceptorDemo;

public static class Inception
{
    [InterceptsLocation(
        filePath: "C:\\_repos\\InterceptorDemo\\Program.cs",
        line: 11,
        column: 8)]
    public static void InterceptSort(
        this ArraySorter arraySorter, int[] array)
    {
        Console.WriteLine($"Intercept using Array.Sort to sort
{array.Length} elements");
        Array.Sort(array);
    }
}
```

You need to tell the interceptor exactly what method to intercept. Let's break this down. We have the following moving parts in our static **`InterceptSort`** method:

- **`filePath`**: The path to your Program.cs file.
- **`line`**: The line in your Program.cs file where **`sorter.Sort(largeArray)`** is found.
- **`column`**: The column number of the start of the **`Sort`** method in the Program.cs file.
- **`this ArraySorter arraySorter`**: The **`InterceptSort`** method is an extension method, so we need to tell it which class to act on (which class we need to intercept)
- **`int[] array`**: Tells our **`InterceptSort`** method that we are passing an array of integers.

To clarify the **`line`** and **`column`** values, have a look at Figure 32.



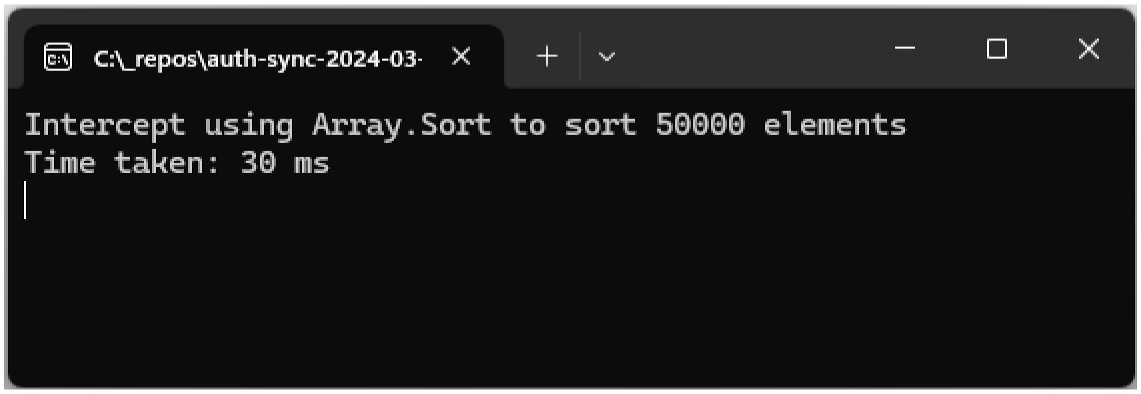
```
using System.Diagnostics;
using InterceptorDemo;

var random = new Random();
var largeArray = Enumerable.Range(0, 50000).Select(x => random.Next()).ToArray();

var sorter = new ArraySorter();

var stopwatch = Stopwatch.StartNew();

sorter.Sort(largeArray);
```

Figure 32: Finding the Line and Column Values

These values can also be found in the bottom-right corner of your editor window in Visual Studio when you place your cursor at the start of the **`Sort`** method.

Note: It goes without saying that the values you will have for your `InterceptSort` ***method for line and column will differ from mine in the code example in Code Listing 50.***

You are now ready to run your application. Give it a whirl and look at the console window seen in Figure 33.



```
Intercept using Array.Sort to sort 50000 elements
Time taken: 30 ms
```

Figure 33: The Sort Method Intercepted

The interceptor jumps into action and grabs the call to the **`Sort`** method, allowing our code to use a more efficient way to sort our array of integers.

For fun, try changing up the number of elements in our array of integers and running the intercepted `Sort` method. Interceptors are really great to work with when thinking about code generators, but how much use they will be to the wider developer community remains to be seen. That is, if interceptors make it out of the experimental phase.

In conclusion

This chapter took us through a buffet of new features introduced in C# 12. One feature that I didn't cover was inline arrays. This is because it is more aimed at the runtime team at Microsoft and library authors. You can read more about inline arrays here.

It is a very niche use case that I didn't think would benefit the larger audience of this book. It's like finding a tub of tofu on the buffet table. Some folks will pick at it, while others won't. Invariably, you will find it largely untouched after the party is over. If I erred in omitting this feature in the book, I apologize. Perhaps it's just me that doesn't like tofu.

That being said, C# 12 brings with it some really interesting features and gives me the feeling that C# 13 might build on some of these in the next release.

In the next chapter, we will have a look at some more new features in .NET 8.

Chapter 3 More New .NET 8 Features

We're not done yet with the new features in .NET 8. There are a few more things we need to talk about that are more suited to a chapter of their own. Starting off this foray into what's new, let us look at AOT.

AOT support

You are now able to compile your code into native code using native AOT. This allows you to take C# and .NET code and compile it to native code the same way C++, Rust, or Go would compile. This means no .NET dependency, no IL code, and no JIT, making for very fast start up times that are very memory efficient.

AOT stands for ahead-of-time and is the counterpart of JIT, which stands for just-in-time compilation.

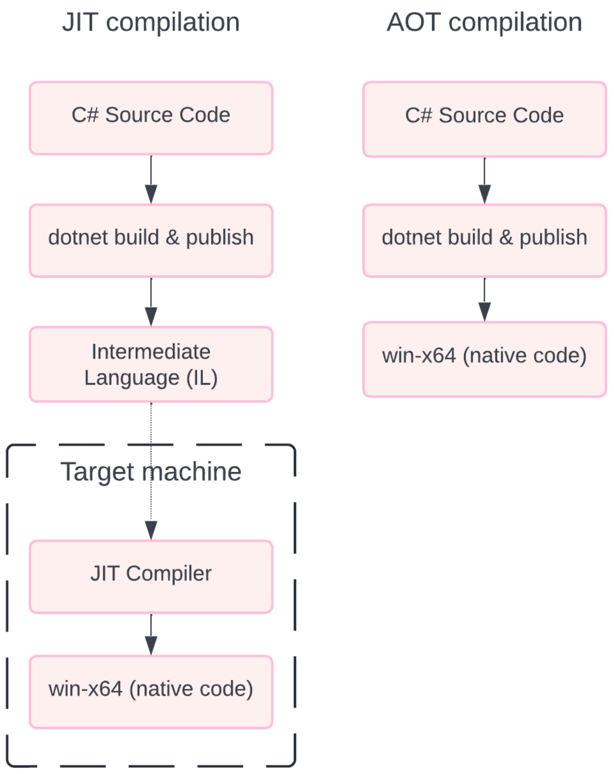
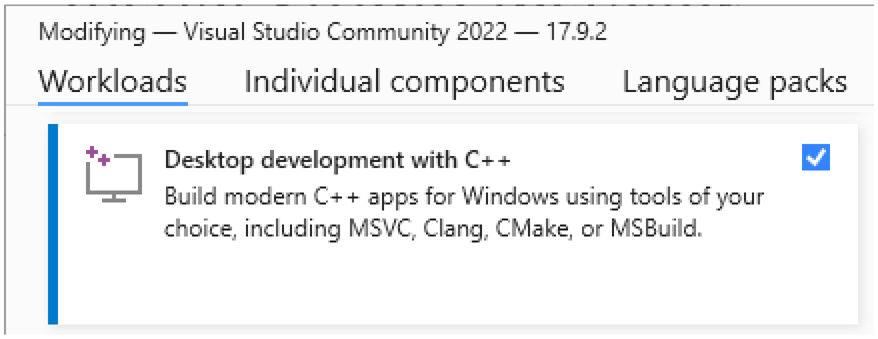


Figure 34: JIT vs. AOT Compilation

From the illustration in Figure 34, you can see how JIT and AOT differ. With AOT, we are generating the native code that runs on the target machine when we compile the application.

With the traditional JIT compiler, the C# code is turned into IL, which is then transformed into native code by the JIT compiler on the target machine.

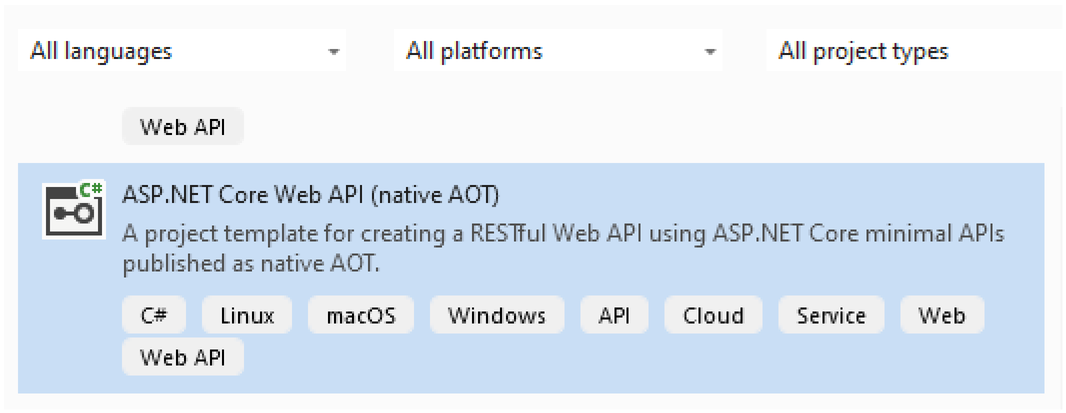
With .NET 7, Native AOT targeted console-type applications, but with .NET 8, ASP.NET Core 8.0 introduced support for AOT.


Modifying — Visual Studio Community 2022 — 17.9.2

Desktop development with C++ ✓
Build modern C++ apps for Windows using tools of your choice, including MSVC, Clang, CMake, or MSBuild.

Figure 35: Prerequisite Workload

A prerequisite for AOT is the addition of the desktop development with the C++ workload. You must have this workload installed before continuing.



ASP.NET Core Web API (native AOT)
A project template for creating a RESTful Web API using ASP.NET Core minimal APIs published as native AOT.

Figure 36: The ASP.NET Core Web API Native AOT Template

Visual Studio now also includes a project template for Native AOT Web APIs. When creating a new AOT Web API project, you will notice a few changes.

Starting with the csproj file, as seen in Code Listing 51, you will notice the inclusion of a **`<PublishAot>`** property in the property group.

This is obviously set to **`true`**.

Code Listing 51: The csproj File

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

  <PropertyGroup>
```

```
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <InvariantGlobalization>true</InvariantGlobalization>
    <PublishAot>true</PublishAot>
  </PropertyGroup>

</Project>
```

Another change that you will notice in Code Listing 52 is the new method **`CreateSlimBuilder`** in the Program.cs file. This registers the minimal number of services required for our API project.

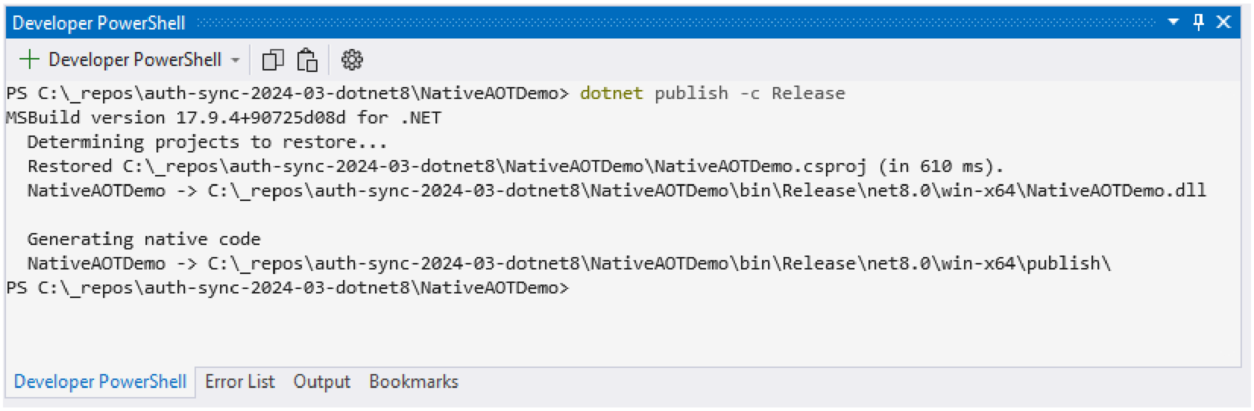
Code Listing 52: The Program.cs File

```
using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);

builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
AppJsonSerializerContext.Default);
});

var app = builder.Build();

var sampleTodos = new Todo[] {
    new(1, "Walk the dog"),
    new(2, "Do the dishes", DateOnly.FromDateTime(DateTime.Now)),
    new(3, "Do the laundry",
DateOnly.FromDateTime(DateTime.Now.AddDays(1))),
    new(4, "Clean the bathroom"),
    new(5, "Clean the car",
DateOnly.FromDateTime(DateTime.Now.AddDays(2)))
};

var todosApi = app.MapGroup("/todos");
todosApi.MapGet("/", () => sampleTodos);
todosApi.MapGet("/{id}", (int id) =>
    sampleTodos.FirstOrDefault(a => a.Id == id) is { } todo
        ? Results.Ok(todo)
        : Results.NotFound());
```

```
app.Run();

public record Todo(int Id, string? Title, DateOnly? DueBy = null, bool
IsComplete = false);

[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{

}
```

You will also notice, by looking at Code Listing 52, that we have a partial class called **`AppJsonSerializerContext`** that is registered with the **`ConfigureHttpJsonOptions`**. This allows the data to be serializable without using reflection and allows us to execute it without using a JIT compiler.

Due to the fact that native code has to be generated, some things, such as reflection-based JSON serialization, will not work with AOT.

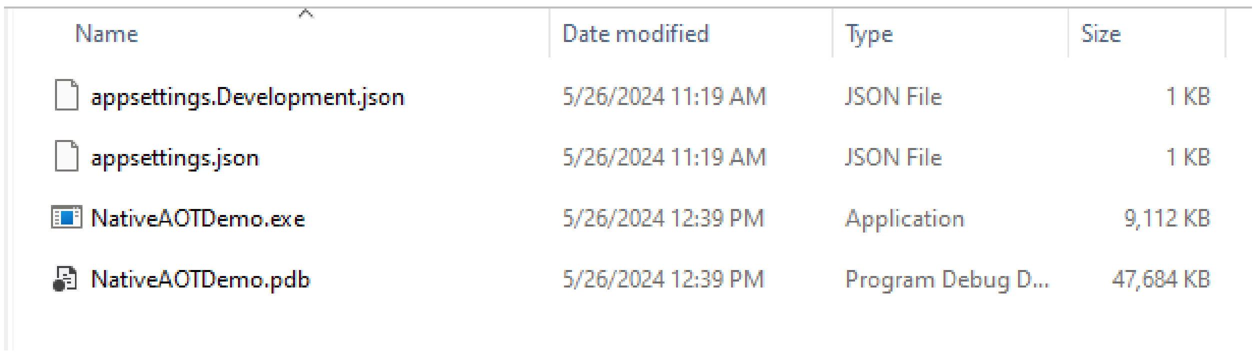


```
PS C:\_repos\auth-sync-2024-03-dotnet8\NativeAOTDemo> dotnet publish -c Release
MSBuild version 17.9.4+90725d08d for .NET
  Determining projects to restore...
  Restored C:\_repos\auth-sync-2024-03-dotnet8\NativeAOTDemo\NativeAOTDemo.csproj (in 610 ms).
  NativeAOTDemo -> C:\_repos\auth-sync-2024-03-dotnet8\NativeAOTDemo\bin\Release\net8.0\win-x64\NativeAOTDemo.dll

  Generating native code
  NativeAOTDemo -> C:\_repos\auth-sync-2024-03-dotnet8\NativeAOTDemo\bin\Release\net8.0\win-x64\publish\
PS C:\_repos\auth-sync-2024-03-dotnet8\NativeAOTDemo>
```

Figure 37: Running dotnet publish

If we have to run **`dotnet publish -c Release`** in the terminal in Visual Studio, you will notice that it generates native code, as seen in Figure 37.

When we look at the published files (see Figure 38), you will notice that it contains a single executable file that is self-contained, because it has been ahead-of-time compiled into native code.



Name	Date modified	Type	Size
appsettings.Development.json	5/26/2024 11:19 AM	JSON File	1 KB
appsettings.json	5/26/2024 11:19 AM	JSON File	1 KB
NativeAOTDemo.exe	5/26/2024 12:39 PM	Application	9,112 KB
NativeAOTDemo.pdb	5/26/2024 12:39 PM	Program Debug D...	47,684 KB

Figure 38: The Published Files

Seeing as Native AOT applications don't use JIT when the application runs, these apps can run in restricted environments where JIT isn't allowed. You can also publish an app using a specific runtime identifier.

Publishing an app for Windows as a native AOT app, you can run **`dotnet publish -r win-x64 -c Release`**.

Publishing an app for Linux as a native AOT app, you can run **`dotnet publish -r linux-arm64 -c Release`**.

Just note that you can't publish cross-platform using AOT. When compiling on Windows, the app is native Windows x64 code. When compiling on Linux, your .exe will not run on Windows.

AOT advantages

AOT has a number of advantages:

- Improvement in startup performance.
- Smaller app size, since it's not JIT.
- Consumes less memory, saving us money when running in the cloud.

AOT disadvantages

AOT also has a few disadvantages:

- As there is no JIT, all native code must be generated at compile time.
- Reflection is not available, which required us to add extra code to serialize JSON data.
- Compile time is longer.
- We require platform-specific tools, such as the C++ tools on Windows.
- No cross-platform publishing means AOT apps are not portable.
- Dependencies must also be AOT-compatible.

When to use AOT

So, how do you know when to use AOT applications? Well, the short answer is: when you create cloud-native APIs. This will allow for increased performance, utilizing fewer resources.

New exception handling in ASP.NET Core 8

Exception handling is something with which all developers are very familiar. When something exceptional happens in your code—something that you didn't expect—you need to handle that. ASP.NET is no exception. While exception handling in ASP.NET was possible before, it required you to write your own middleware in order to handle it correctly.

With .NET 8, Microsoft has now given developers the ability to have an exception handler class, specifically made for exception handling, that you can add to your pipeline. To illustrate this, I will create a very simple Web API project, as seen in Code Listing 53.

Code Listing 53: A Basic Web API Project

```
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();


app.MapGet("/api/gettasks", () =>
{
    throw new InvalidProgramException("Something went wrong");
});

app.Run();
```

I am simply throwing an exception when the **`/gettasks`** endpoint is called. I also have a .http file that allows me to test my API inside Visual Studio. The code is illustrated in Code Listing 54.

Code Listing 54: The .http File

```
@ExceptionHandlingDemo_HostAddress = http://localhost:5105

GET {{ExceptionHandlingDemo_HostAddress}}/api/gettasks
Accept: application/json

###
```

Your port might be different from mine, but you can configure this in the debug properties if you like. Speaking of the debug properties, you can find these under the **Debug** menu in Visual Studio, as seen in Figure 39.

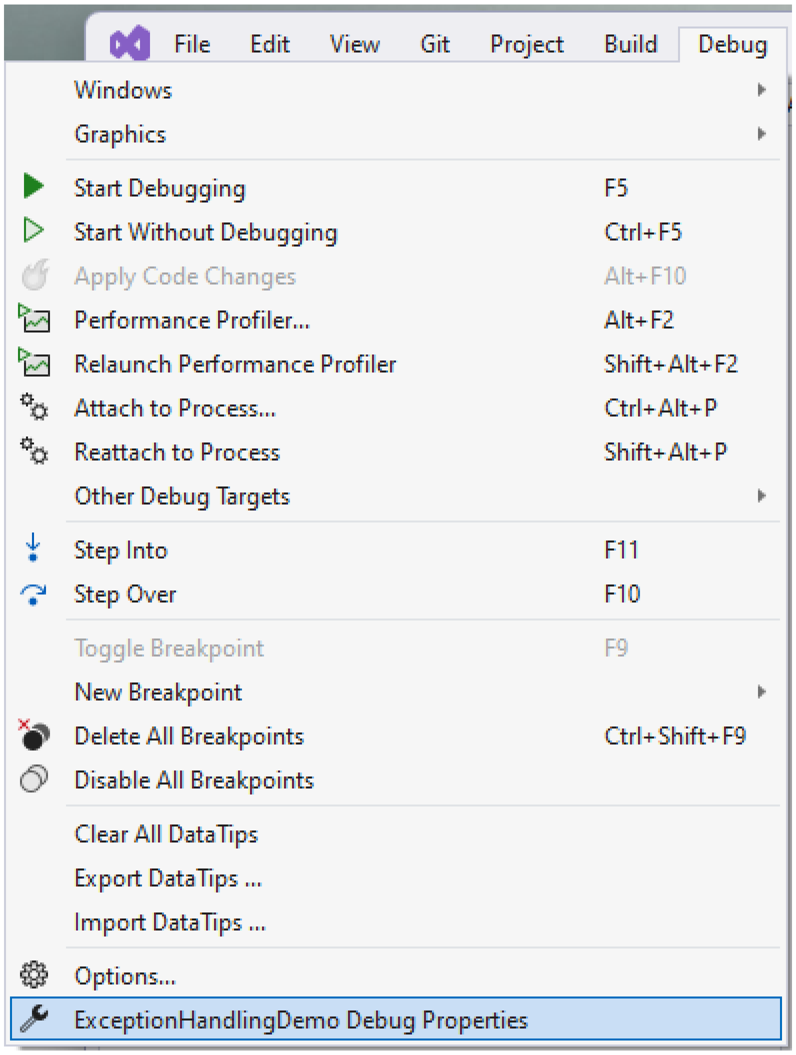
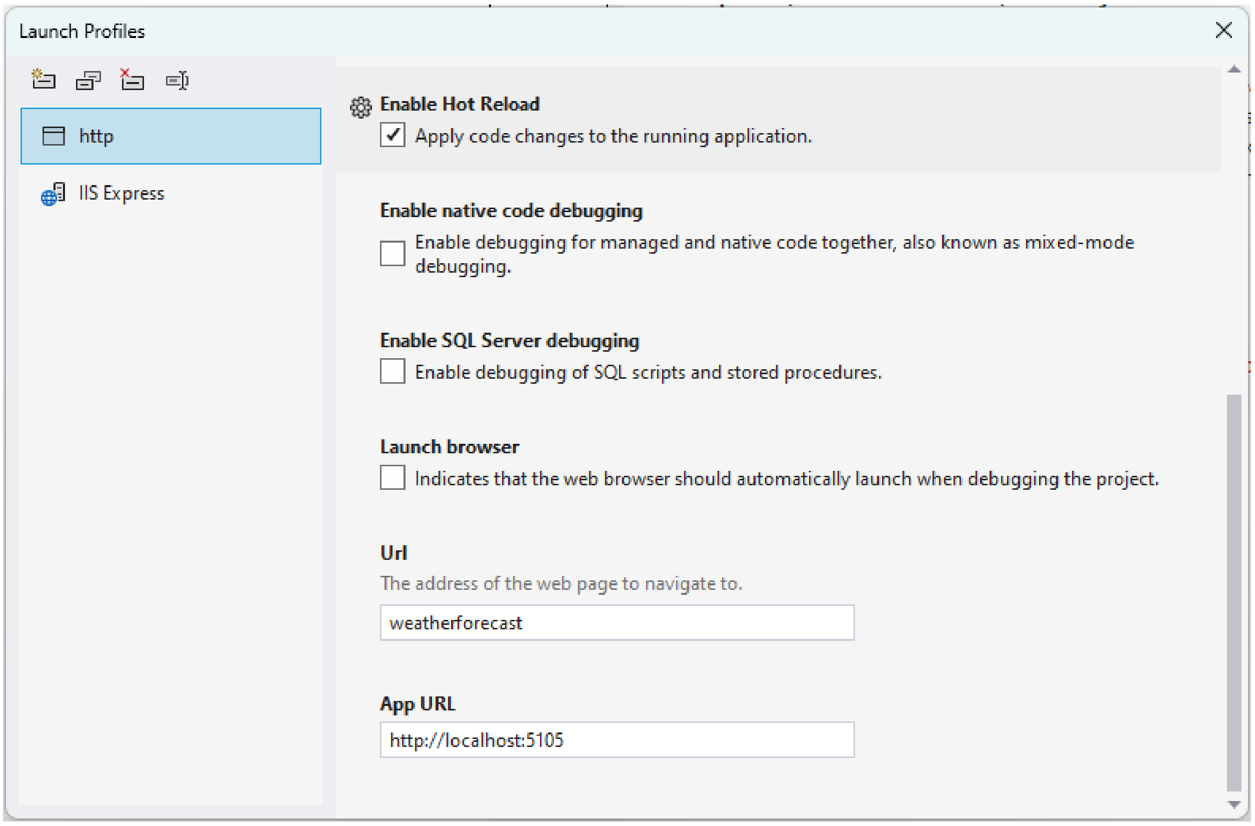

File
Edit
View
Git
Project
Build
Debug
Windows
Graphics
Start Debugging F5
Start Without Debugging Ctrl+F5
Apply Code Changes Alt+F10
Performance Profiler... Alt+F2
Relaunch Performance Profiler Shift+Alt+F2
Attach to Process... Ctrl+Alt+P
Reattach to Process Shift+Alt+P
Other Debug Targets
Step Into F11
Step Over F10
Toggle Breakpoint F9
New Breakpoint
Delete All Breakpoints Ctrl+Shift+F9
Disable All Breakpoints
Clear All DataTips
Export DataTips ...
Import DataTips ...
Options...
ExceptionHandlingDemo Debug Properties

Figure 39: Finding the Debug Properties

Clicking on the menu called **Debug Properties** will display the Launch Profiles window.

Note: It seems strange that a menu called Debug Properties would launch a window called Launch Profiles. Just be aware of this disconnect in the naming. It might be a bug in Visual Studio.

As seen in Figure 40, the Launch Profiles window allows you to specify the app URL. You can also uncheck the Launch browser option (as I have done) to run the API without opening a browser window.


Launch Profiles

http

IIS Express

Enable Hot Reload

✓ Apply code changes to the running application.

Enable native code debugging

☐ Enable debugging for managed and native code together, also known as mixed-mode debugging.

Enable SQL Server debugging

☐ Enable debugging of SQL scripts and stored procedures.

Launch browser

☐ Indicates that the web browser should automatically launch when debugging the project.

Url

The address of the web page to navigate to.

weatherforecast

App URL

http://localhost:5105

Figure 40: Launch Profiles

Running my API and sending a request to my **`/gettasks`** endpoint results in the exception, as seen in Figure 41.

This response window is quite rich in the information it returns. You can see the raw response, the headers, and the request that resulted in the exception.

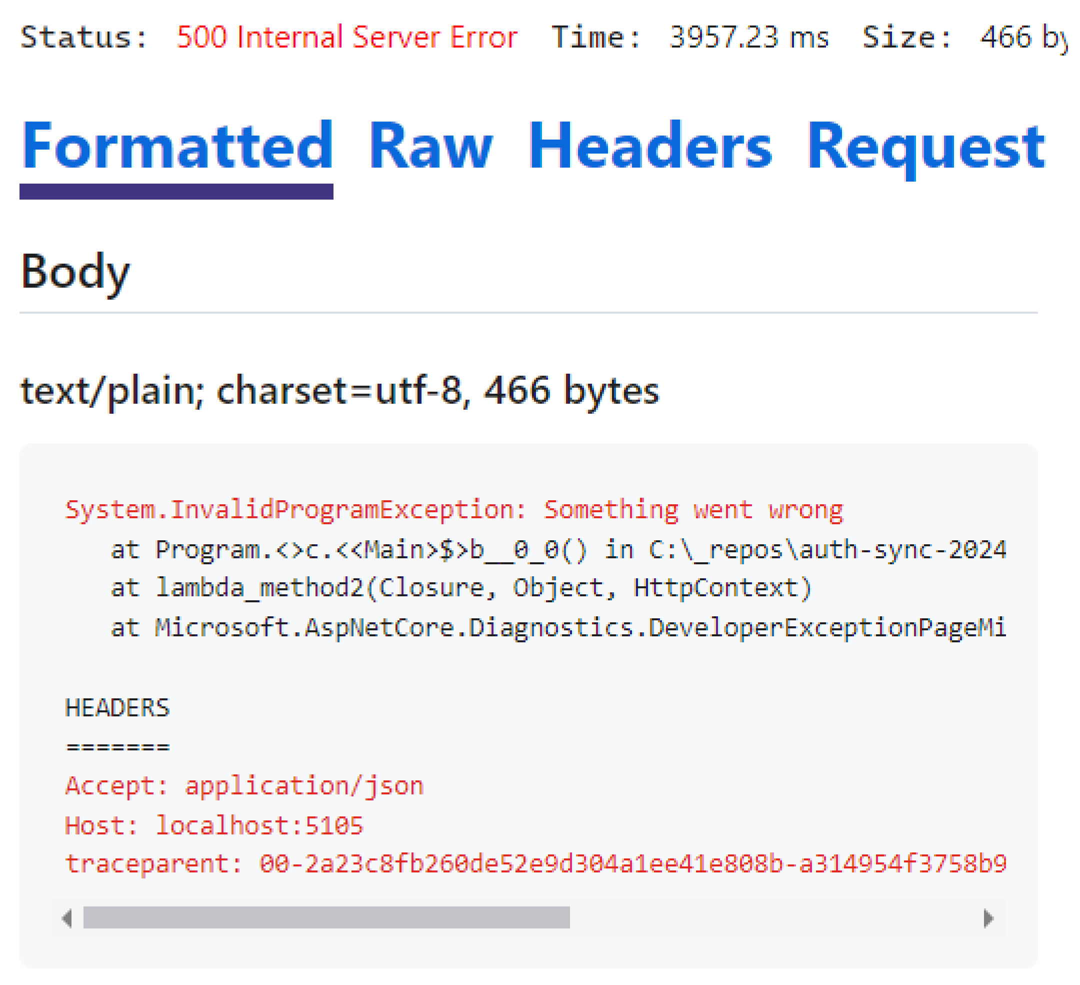
But while the default is fine, we might want to handle exceptions on our own—and this is where the exception handler class comes into play.

`IExceptionHandler` is a new interface that allows developers to handle exceptions from a known location. Implementations of **`IExceptionhandler`** are registered by calling **`IServiceCollection.AddExceptionHandler<T>`**.

If you register multiple implementations, they are called in the order they're registered. Once an exception handler handles a request, you can return **`true`** to stop processing, or **`false`** to continue processing.

Any exceptions not handled by the exception handler will then fall back to the default behavior from middleware.

Let's see how to implement this in our project by looking at some code.


Status: 500 Internal Server Error Time: 3957.23 ms Size: 466 by

Body

text/plain; charset=utf-8, 466 bytes

```
System.InvalidProgramException: Something went wrong
   at Program.<>c.<<Main>$>b__0_0() in C:\_repos\auth-sync-2024
   at lambda_method2(Closure, Object, HttpContext)
   at Microsoft.AspNetCore.Diagnostics.DeveloperExceptionPageMi

HEADERS
=======
Accept: application/json
Host: localhost:5105
traceparent: 00-2a23c8fb260de52e9d304a1ee41e808b-a314954f3758b9
```

Figure 41: The Request Exception

Have a look at the code in Code Listing 55. This gives us a lot of control over the exceptions we handle in our application. You will notice that I simply return **`true`**, which will stop the processing right there.

Code Listing 55: Our CustomExceptionHandler Class Implementing IExceptionHandler

```
using Microsoft.AspNetCore.Diagnostics;

namespace ExceptionHandlingDemo;
public class CustomExceptionHandler : IExceptionHandler
{
    public async ValueTask<bool> TryHandleAsync(HttpContext httpContext,
        Exception ex,
        CancellationToken cancellationToken)
    {
        httpContext.Response.StatusCode = 500;
        httpContext.Response.ContentType = "text/plain";
        await httpContext.Response.WriteAsync($"Custom Exception Handler: 
{ex.Message}");

        return true;
    }
}
```

Now that we have our custom exception handler, we need to add it to our services, as seen in Code Listing 56.

Code Listing 56: Modifying the Program.cs File

```
using ExceptionHandlingDemo;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddExceptionHandler<CustomExceptionHandler>();

var app = builder.Build();
app.UseExceptionHandler(_ => { });

app.MapGet("/api/gettasks", () =>
{
    throw new InvalidProgramException("Something went wrong");
});

app.Run();
```

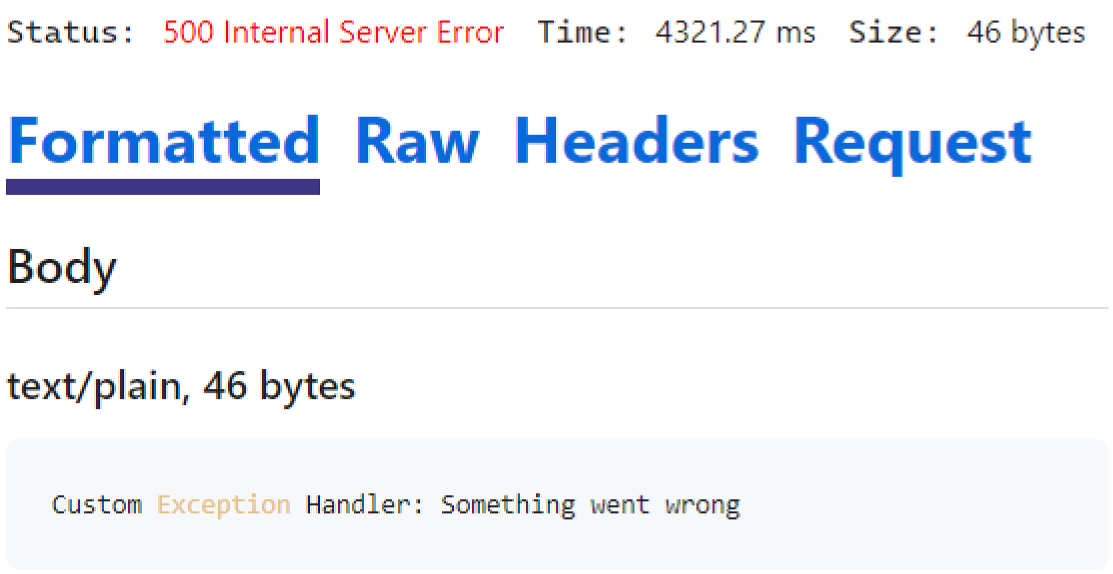
You will notice that **`AddExceptionHandler`** is a new API, and we just pass it the **`CustomExceptionHandler`** class. In effect, this adds our **`IExceptionHandler`** implementation to the services. These implementations are then used by the exception handler middleware to handle unexpected exceptions.

We then add the middleware by adding the code **`app.UseExceptionHandler(_ => { });`** to the Program.cs file. You should notice the **`_ => {}`** configuration lambda here; this is because the middleware is supposed to look for the **`CustomExceptionHandler`** we created, but it doesn't. If you used **`app.UseExceptionHandler();`**, you will immediately see an error when running your application. This is because **`UseExceptionHandler`** doesn't assume that if we have **`AddExceptionHandler`**, that it must use our custom **`IExceptionHandler`** implementation.

You can view the discussion on GitHub here.

As a workaround for this error, we just use the empty configuration lambda using the _ discards. If we wanted to, we could configure options here for our custom exception handler. For this example, however, we just pass in the empty configuration lambda.

Running the API again and calling our **`/gettasks`** endpoint, we will see the exception handled by our custom exception handler, as expected (see Figure 42).


Status: 500 Internal Server Error   Time: 4321.27 ms   Size: 46 bytes

text/plain, 46 bytes

```
Custom Exception Handler: Something went wrong
```

Figure 42: The Custom Exception Handler in Action

And that's all there is to it. This is the new exception handling capability in ASP.NET Core 8, allowing developers to streamline and enhance error management within their applications. Exception handling is critical to any robust application. In ASP.NET Core 8, **`IExceptionHandler`** improves this significantly. It allows for cleaner, more maintainable code.

Bearer tokens in .NET 8 Identity

Thinking about modern .NET applications, identity and auth in .NET have been, to put it mildly, somewhat painful. In .NET 8, however, Microsoft set out to improve identity and auth. Suffice it to say, it has gotten a lot better in .NET 8.

Code Listing 57: A Basic Web API

```
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.Run();
```

Looking at Code Listing 57, we will start off with the simplest code possible.

We are going to turn something really simple into the code in Code Listing 58, which has identity and auth in it.

Code Listing 58: The Completed Web API Code

```
using System.Security.Claims;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthentication()
    .AddBearerToken(IdentityConstants.BearerScheme);
builder.Services.AddAuthorizationBuilder();

builder.Services.AddDbContext<AppDbContext>(options =>
options.UseSqlite("Datasource=adminapp.db"));

builder.Services.AddIdentityCore<AdminUser>()
    .AddEntityFrameworkStores<AppDbContext>()
    .AddApiEndpoints();

var app = builder.Build();

app.MapIdentityApi<AdminUser>();


app.MapGet("/", (ClaimsPrincipal user) => $"Welcome
{user.Identity!.Name}")
    .RequireAuthorization();

app.Run();


class AdminUser : IdentityUser { }

class AppDbContext : IdentityDbContext<AdminUser>
{
    public AppDbContext(DbContextOptions<AppDbContext> options) :
base(options) { }
}
```

Let’s start breaking this down. You will notice the following code, **`class AdminUser : IdentityUser { }`**, which is just my user that extends the **`IdentityUser`** class.

The next thing I’m going to do is wire up some services.

As seen in Code Listing 59, I have an extract of the code in Code Listing 58. Here, I am starting off by adding **`AddAuthentication()`**, which registers services required by authentication services, and then adding **`AddBearerToken`**, which adds bearer token authentication.

Note that this is not a JSON web token, but it is still a self-contained, stateless bearer token.

Next, we add **`AddAuthorizationBuilder()`**, which adds authorization services to the **`IServiceCollection`**. From an auth perspective, that's all I needed to do.

Code Listing 59: Wiring Up Services

```
builder.Services.AddAuthentication()
    .AddBearerToken(IdentityConstants.BearerScheme);
builder.Services.AddAuthorizationBuilder();
```

To store the user data, I will be using Entity Framework Core and SQLite as the provider. In .NET 8, Entity Framework is really fast, a perfectly viable solution for your requirements. First, we need to add some NuGet packages:

- Microsoft.AspNetCore.Identity.EntityFrameworkCore
- Microsoft.EntityFrameworkCore.Sqlite
- Microsoft.EntityFrameworkCore.Design

We add the design NuGet package because we are going to be working with migrations. We now need to add a **`DbContext`**, which we add as seen in Code Listing 60, extending the **`IdentityDbContext`** class using the **`AdminUser`** as the user.

Code Listing 60: Adding the DB Context

```
class AppDbContext : IdentityDbContext<AdminUser>
{
    public AppDbContext(DbContextOptions<AppDbContext> options) :
base(options) { }
}
```

In Code Listing 61, we register the **`DbContext`** to use SQLite as the provider, with a data source called **`adminapp.db`**.

Code Listing 61: Registering DB Context Services

```
builder.Services.AddDbContext<AppDbContext>(options =>
options.UseSqlite("Datasource=adminapp.db"));

builder.Services.AddIdentityCore<AdminUser>()
    .AddEntityFrameworkStores<AppDbContext>()
    .AddApiEndpoints();
```

After registering the **`DbContext`**, we add identity with **`AddIdentityCore`**, using **`AdminUser`**. We also add **`AddEntityFrameworkStores()`**, which adds the Entity Framework implementation of identity information stores.

We then use **`AddApiEndpoints()`** to add configuration and services required to support **`IdentityApiEndpointRouteBuilderExtensions`**.

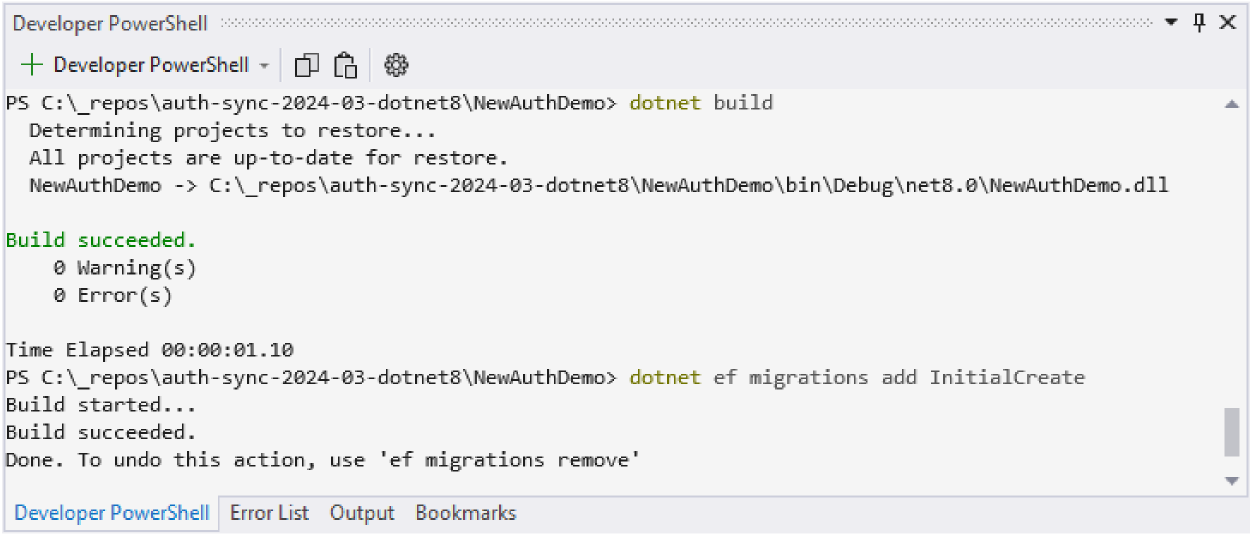
Code Listing 62 illustrates the game-changing code in this project of ours. Before .NET 8, if you wanted to have registration endpoints, login endpoints, or refresh token endpoints, you had to manually map these. That is no longer required. Now, all you need to do is call **`MapIdentityApi`** and specify the user object—and that's it.

Code Listing 62: Adding MapIdentityApi

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

app.MapIdentityApi<AdminUser>();
app.MapGet("/", (ClaimsPrincipal user) => $"Welcome
{user.Identity!.Name}")
    .RequireAuthorization();

app.Run();
```

This will go ahead and add all those endpoints for you. To illustrate an endpoint that requires authorization, I am adding authorization to the root that simply returns a welcome message to an authenticated user.

To see this in action, we need to run our migrations. Do this by running **`dotnet build`** to check if the build succeeds and then run **`dotnet ef migrations add InitialCreate`** to add the migrations, as seen in Figure 43.

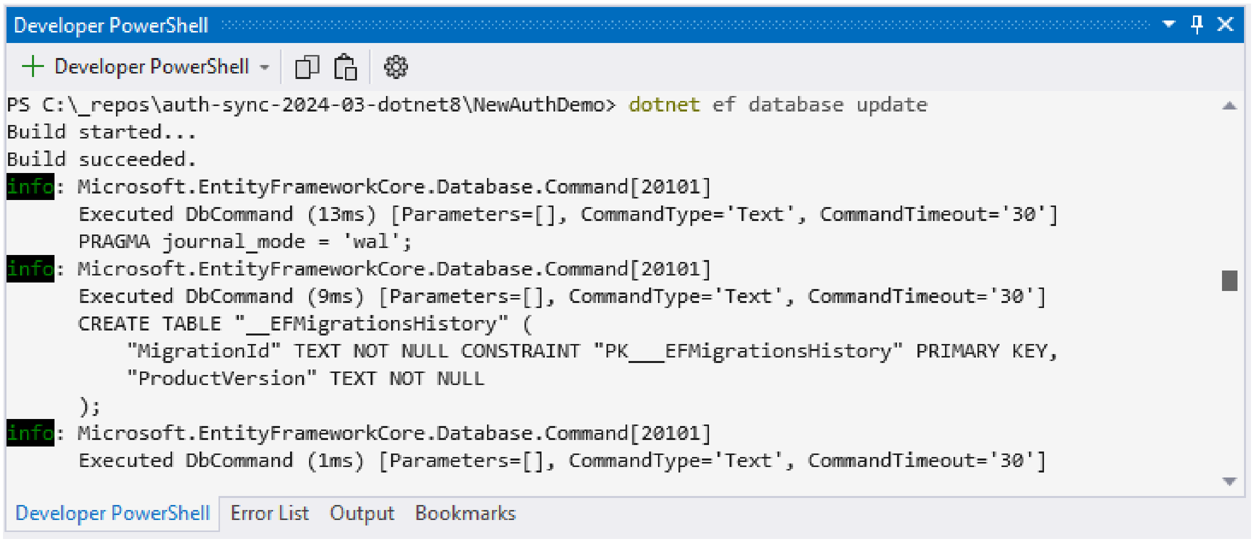


```
PS C:\_repos\auth-sync-2024-03-dotnet8\NewAuthDemo> dotnet build
  Determining projects to restore...
  All projects are up-to-date for restore.
  NewAuthDemo -> C:\_repos\auth-sync-2024-03-dotnet8\NewAuthDemo\bin\Debug\net8.0\NewAuthDemo.dll

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:01.10
PS C:\_repos\auth-sync-2024-03-dotnet8\NewAuthDemo> dotnet ef migrations add InitialCreate
Build started...
Build succeeded.
Done. To undo this action, use 'ef migrations remove'
```

Figure 43: Adding Migrations

If you receive an error message stating, *Could not execute because the specified command or file was not found*, try running **`dotnet tool install --global dotnet-ef`**. After that has completed, try running **`dotnet ef migrations add InitialCreate`** again.

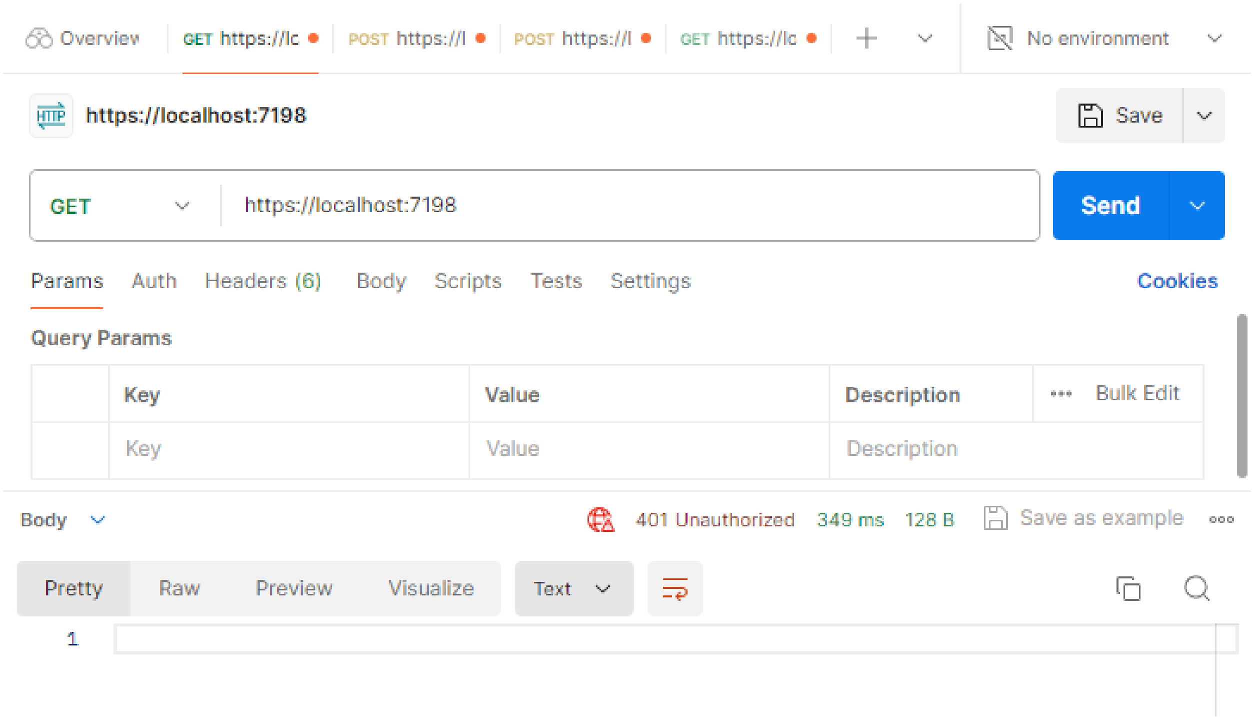
Lastly, to create your SQLite file, run **`dotnet ef database update`**, as seen in Figure 44.



```
PS C:\_repos\auth-sync-2024-03-dotnet8\NewAuthDemo> dotnet ef database update
Build started...
Build succeeded.
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (13ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      PRAGMA journal_mode = 'wal';
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (9ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      CREATE TABLE "__EFMigrationsHistory" (
          "MigrationId" TEXT NOT NULL CONSTRAINT "PK___EFMigrationsHistory" PRIMARY KEY,
          "ProductVersion" TEXT NOT NULL
      );
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (1ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
```

Figure 44: Creating the SQLite File

We are now ready to run our API and test the endpoints using Postman.



https://localhost:7198

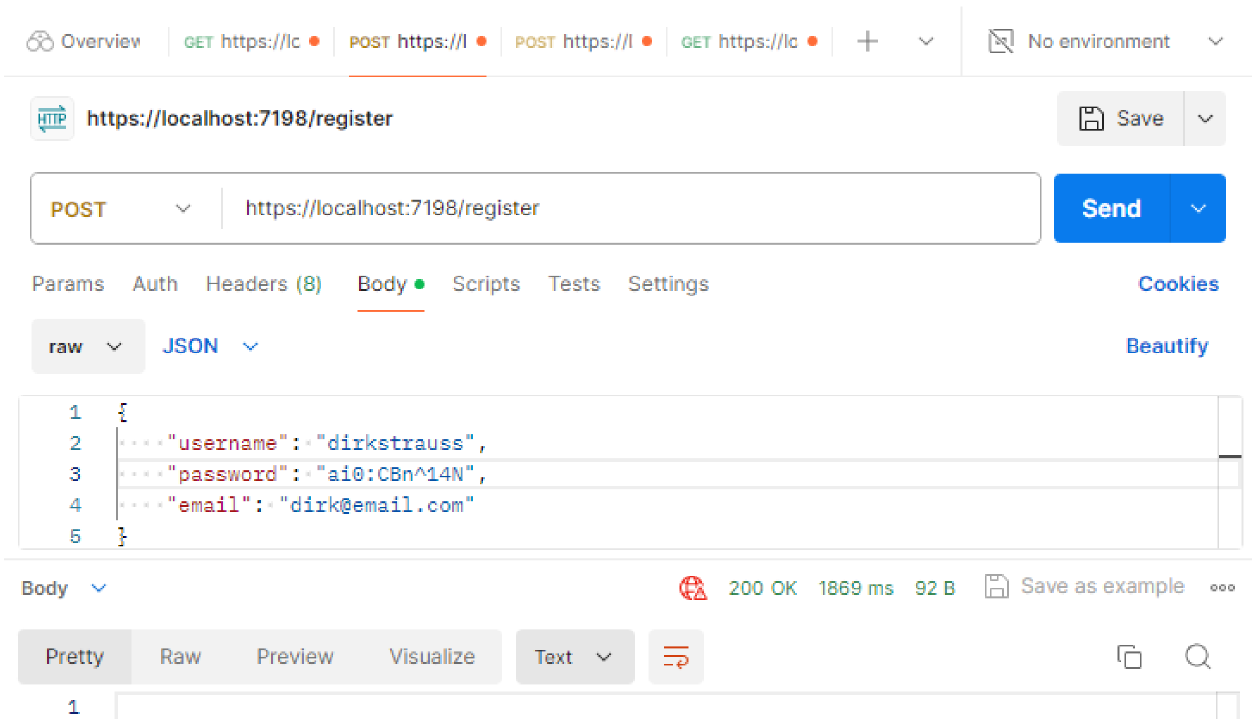
GET https://localhost:7198

	Key	Value	Description
	Key	Value	Description

401 Unauthorized 349 ms 128 B

Figure 45: Trying to Access API Unauthorized

Looking at Figure 45, if I try to call the API, I will get a **`401 Unauthorized`** response. I have to obtain a bearer token first, and in order to get a bearer token, I need to register.



```
{
    "username": "dirkstrauss",
    "password": "ai0:CBn^14N",
    "email": "dirk@email.com"
}
```

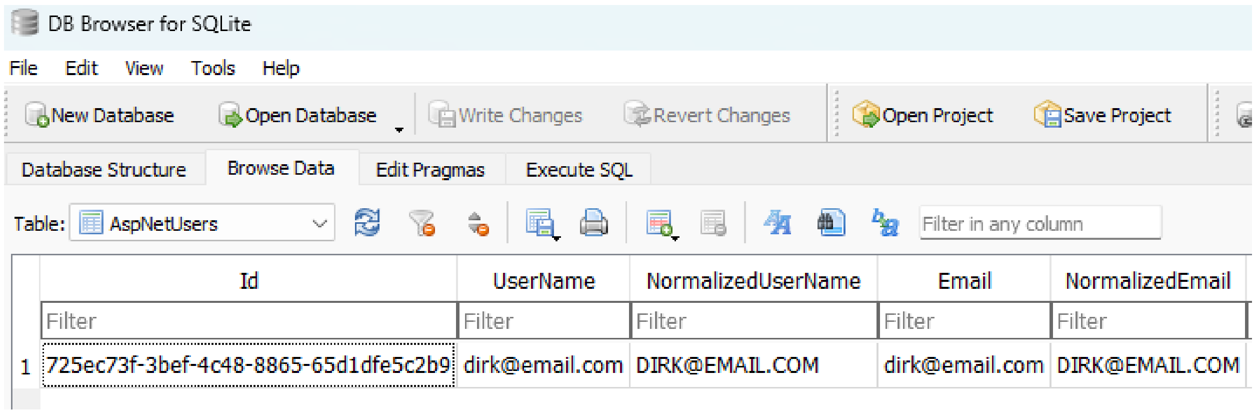
Figure 46: Calling the Register Endpoint

As seen in Figure 46, call the **`/register`** endpoint using the JSON in Code Listing 63.

Code Listing 63: The Register JSON

```
{
    "username": "dirkstrauss",
    "password": "ai0:CBn^14N",
    "email": "dirk@email.com"
}
```

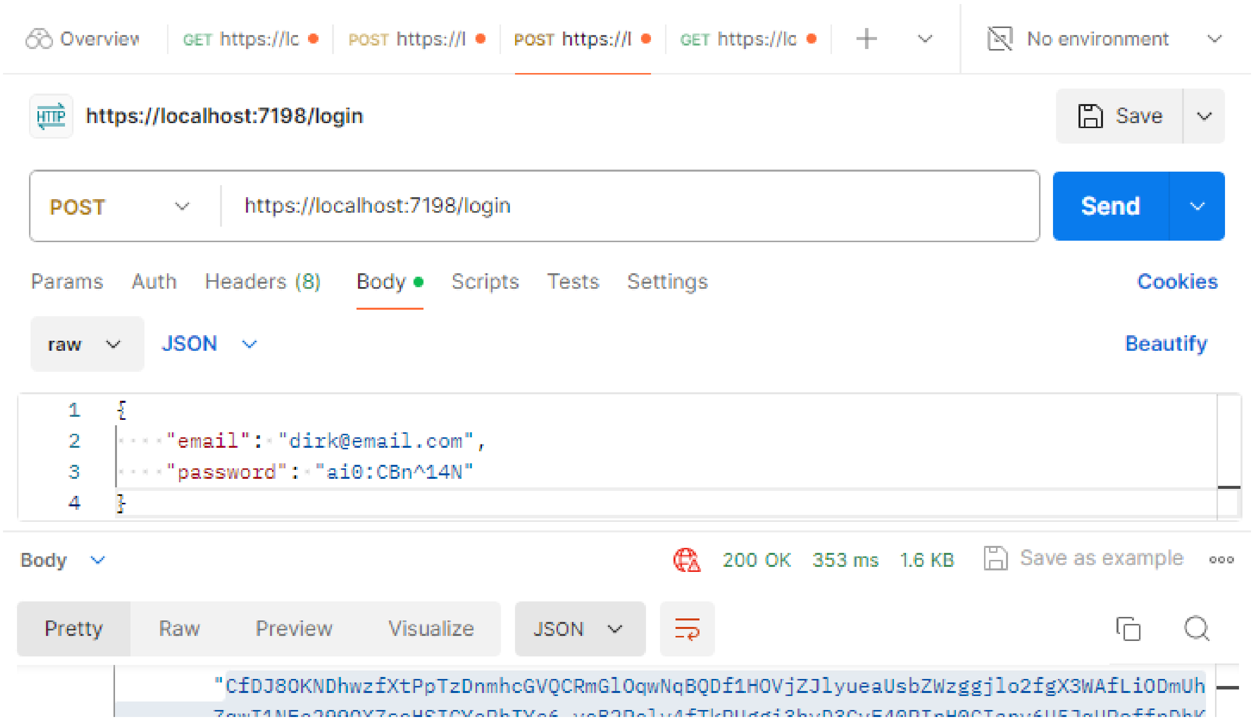
This will register me here as a user in my API. Looking at my SQLite table in Figure 47, I can see that my user has been registered successfully with all the information provided.


DB Browser for SQLite
File Edit View Tools Help
New Database Open Database Write Changes Revert Changes Open Project Save Project
Database Structure Browse Data Edit Pragmas Execute SQL
Table: AspNetUsers
Filter in any column

	Id	UserName	NormalizedUserName	Email	NormalizedEmail
1	725ec73f-3bef-4c48-8865-65d1dfe5c2b9	dirk@email.com	DIRK@EMAIL.COM	dirk@email.com	DIRK@EMAIL.COM

Figure 47: The Registered User

After registering, I can attempt to obtain a bearer token from my **`/login`** endpoint, as seen in Figure 48.



https://localhost:7198/login

POST https://localhost:7198/login

```
{
    "email": "dirk@email.com",
    "password": "ai0:CBn^14N"
}
```

```
"CfDJ8OKNDhwzfXtPpTzDnmhcGVQCRmGlOqwNqBQDf1HOVjZJlyueaUsbZWzggjlo2fgX3WAfLiODmUh
```

Figure 48: Calling the Login Endpoint

If my username and password are correct, I will receive a **`200 OK`** response, and I will receive a bearer token and a refresh token, as seen in Code Listing 64.

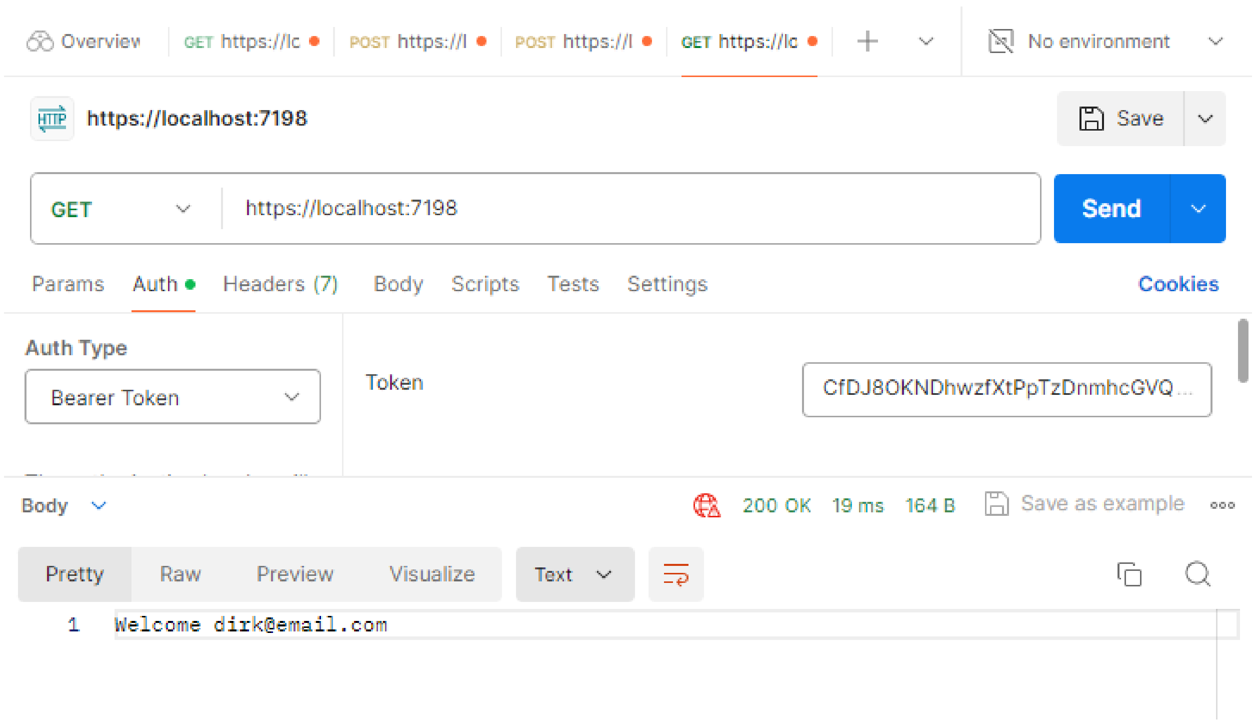
Code Listing 64: The Login Response

```
{
    "tokenType": "Bearer",
    "accessToken": "CfDJ8OKNDhwzfXtPpTzDnmhcGVTqch0Me2U8UB_bc-wd-H2Xg",
    "expiresIn": 3600,
    "refreshToken": "012QZDywGuk3qP-eSZ8apG0D4DEg06tcykxqH16-KiEs7dgg"
}
```

I can use the bearer token in the next step.

Note: The bearer token and refresh token in Code Listing 64 have been edited to be shorter. This was to display the response neatly in the code block. In reality, the tokens returned are much longer.

I then try to call my root endpoint again, only this time, I add the bearer token to the **Auth** tab in Postman. As seen in Figure 49, I received a successful welcome response back.



```
Welcome dirk@email.com
```

Figure 49: Calling the Root Endpoint Again

It is also important to note that when calling the endpoint using the bearer token, I am not making any calls back to the database to validate that token. This is because it is a stateless, self-contained token.

The last thing I want to touch on is the refresh token. The purpose of the refresh token in Code Listing 64 is to be stored locally on the client. It can then be used when the bearer token has expired.

This means that I can go to the `/refresh` endpoint and do an `HTTP POST` using the JSON in Code Listing 65.

Code Listing 65: Calling the Refresh API

```
{
  "refreshToken": "012QZDywGuk3qP-eSZ8apG0D4DEg06tcykxqH16-KiEs7dgg"
}
```

If it is a valid refresh token, then it will return a new bearer and refresh token for me. If you are interested in seeing what endpoints are available, place your mouse cursor on **`MapIdentityApi`** and press **`Ctrl+F12`**.

You will see the following endpoints:

- **`/register`**
- **`/login`**
- **`/refresh`**
- **`/confirmEmail`**
- **`/resendConfirmationEmail`**
- **`/forgotPassword`**
- **`/resetPassword`**
- **`/manage/2fa`**
- **`/manage/info`**

This makes it extremely easy to set up authorization and identity on any app. This is not exclusive to minimal web APIs.

Data annotations updates

There are a few new data annotation attributes added to .NET 8. As you know, data annotations are used mainly for validation purposes, and in this example, I will be using an API to call endpoints that validate the models in my project.

As seen in Code Listing 66, I have a controller that calls out to a few APIs. The model is validated, and if valid, just returns an **`Ok`** result.

If, however, the model is invalid, it means that one of the validations applied by the data annotation attributes caught an invalid input.

We will be looking at the following data annotations:

- **`Range`**
- **`Length`**
- **`AllowedValues`**
- **`DeniedValues`**
- **`Base64`**

I am also going to use an .http file to test my API endpoints.

Code Listing 66: The Demo Controller

```
using DataAnnotationsDemo.Models;
using Microsoft.AspNetCore.Mvc;

namespace DataAnnotationsDemo.Controllers;

[ApiController]
[Route("[controller]")]
public class DemoController : Controller
{
    [HttpPost("rangedemo")]
    public IActionResult RangeDemo([FromBody] RangeExampleModel body)
        => !ModelState.IsValid ? BadRequest(ModelState) : Ok(body);

    [HttpPost("lengthdemo")]
    public IActionResult LengthDemo([FromBody] LengthExampleModel body)
        => !ModelState.IsValid ? BadRequest(ModelState) : Ok(body);

    [HttpPost("allowedvaluesdemo")]
    public IActionResult AllowedValuesDemo([FromBody]
AllowedValuesExampleModel body)
        => !ModelState.IsValid ? BadRequest(ModelState) : Ok(body);

    [HttpPost("deniedvaluesdemo")]
    public IActionResult DeniedValuesDemo([FromBody]
DeniedValuesExampleModel body)
        => !ModelState.IsValid ? BadRequest(ModelState) : Ok(body);

    [HttpPost("base64demo")]
    public IActionResult Base64Demo([FromBody] Base64ExampleModel body)
        => !ModelState.IsValid ? BadRequest(ModelState) : Ok(body);
}
```

Code Listing 67 shows the code to test the various endpoints. The code in the following .http file will all validate successfully. Therefore, if you run the project and call these endpoints, you will receive an `Ok` result.

Code Listing 67: The .http File

```
@DataAnnotationsDemo_HostAddress = http://localhost:5053

POST {{DataAnnotationsDemo_HostAddress}}/demo/rangedemo
Content-Type: application/json
{
  "threshold": 50
}

###
POST {{DataAnnotationsDemo_HostAddress}}/demo/lengthdemo
Content-Type: application/json
{
  "testresults": [
        2,
        5,
        7
  ]
}

###
POST {{DataAnnotationsDemo_HostAddress}}/demo/allowedvaluesdemo
Content-Type: application/json
{
  "countryiso": "USA"
}


###
POST {{DataAnnotationsDemo_HostAddress}}/demo/deniedvaluesdemo
Content-Type: application/json
{
  "countryiso": "USA"
}


###
POST {{DataAnnotationsDemo_HostAddress}}/demo/base64demo
Content-Type: application/json
{
  "systeminput": "rtdfghjuytrfvcxs8796"
}
```

Range attribute with minimum and maximum exclusive

The **`Range`** attribute now allows you to set exclusive bounds on the range being validated. From the code in Code Listing 68, you will see that I have a range of **`20`** to **`80`** set with the **`MinimumIsExclusive`** and **`MaximumIsExclusive`** set to **`true`**.

What this means is that I do not want to accept **`20`** as the lower value in the range, and I do not want to accept **`80`** as my upper value. The valid values are, therefore, any values between **`20`** and **`80`**, excluding **`20`** and **`80`**.

This is why **`{ "threshold": 50 }`** will validate, while **`{ "threshold": 20 }`** and any value lower, as well as **`{ "threshold": 80 }`** and any value higher, will fail validation.

If you wanted to include **`20`** but exclude **`80`**, you would set **`MinimumIsExclusive = false`** while keeping **`MaximumIsExclusive`** unchanged.

Code Listing 68: Range Attribute with Minimum and Maximum Validation

```
using System.ComponentModel.DataAnnotations;

namespace DataAnnotationsDemo.Models;

public class RangeExampleModel
{
    [Range(20, 80, MinimumIsExclusive = true, MaximumIsExclusive = true)]
    public int Threshold { get; set; }
}
```

Simple and effective. I'm not sure how much mileage this attribute will get, but it is nice to have the option to include or exclude the lower or upper bounds of the range values.

Length attribute

Keeping in the same vein when it comes to validation attributes, we now have a **`Length`** attribute, as seen in Code Listing 69.

Code Listing 69: Length Validation Attribute

```
using System.ComponentModel.DataAnnotations;

namespace DataAnnotationsDemo.Models;

public class LengthExampleModel
{
    [Length(1, 3)]
    public ICollection<int> TestResults { get; set; } =
Array.Empty<int>();
}
```

Here I want to validate the length of my collection of test results. I no longer need to chain **`MinLength`** and **`MaxLength`**, for example **`[MinLength(1), MaxLength(3)]`**, to make this work. I can now simply state **`[Length(1, 3)]`** and be done with it. Also note that the parameters for the **`Length(a, b)`** attribute are always inclusive.

AllowedValues attribute

As seen in Code Listing 70, you've guessed it, we have the **`AllowedValues`** attribute. This allows you to be explicit with which values you will allow.

Code Listing 70: Allowed Values Attribute

```
using System.ComponentModel.DataAnnotations;

namespace DataAnnotationsDemo.Models;

public class AllowedValuesExampleModel
{
    [AllowedValues("USA", "GBR", "ZAF")]
    public string CountryISO { get; set; } = string.Empty;
}
```

This means that the API will only accept the following values:

- **`{ "countryiso": "USA" }`**
- **`{ "countryiso": "GBR" }`**
- **`{ "countryiso": "ZAF" }`**

Anything else will fail validation, for example, passing **`ZAR`** instead of **`ZAF`**. **`ZAR`** is the currency code for South Africa, while **`ZAF`** is the ISO country code.

DeniedValues attribute

Going hand-in-hand with **`AllowedValues`** is the **`DeniedValues`** attribute. As seen in Code Listing 71, you can be just as explicit with values that you do not want to allow.

Code Listing 71: Denied Values Attribute

```
using System.ComponentModel.DataAnnotations;

namespace DataAnnotationsDemo.Models;

public class DeniedValuesExampleModel
{
    [DeniedValues("US", "ZAR")]
    public string CountryISO { get; set; } = string.Empty;
}
```

This is very handy for a variety of applications. In this example, only, **`{ "countryiso": "US" }`** and **`{ "countryiso": "ZAR" }`** will fail validation. Everything else is permitted.

Base64String attribute

The last validation attribute is the **`Base64String`** attribute. This validates that the string passed is a valid Base64 representation.

Code Listing 72: Base64 String Validation Attribute

```
using System.ComponentModel.DataAnnotations;

namespace DataAnnotationsDemo.Models;

public class Base64ExampleModel
{
    [Base64String]
    public string SystemInput { get; set; } = string.Empty;
}
```

A point to note here is that this does not determine that a string is a valid Base64 string that can be decoded into something. In other words, if we decode **`SGVsbG8gd29ybGQ=`**, we will get **`Hello world`**. If, however, if we try to decode **`rtdfghjuytrfvcxs8796`**, it will decode to gibberish.

All that the **`Base64String`** attribute does is check that the string contains valid Base64 characters (A-Z, a-z, 0-9, +,/,=).

In conclusion

.NET 8 brings with it so many new features and enhancements, it’s difficult to collect them all in a single book. We haven’t even started looking in depth at the performance improvements in .NET 8. That alone is significant.

If you want to read more about the topic of performance, have a look at this article by Stephen Toub.

With the advent of AI and the expectation of AGI soon afterwards, you might be wondering if your profession is in danger. I take solace in the fact that AI currently needs specific instructions in order to deliver the best results.

As developers, we all know that walking on water and developing an application perfectly from a customer's specification is possible if both are frozen. While this might be true for now, with AI serving as a supplementary assistant to humans, this dynamic may soon change.

We need to remember that AI is but a machine capable of extrapolating the most likely answer based on terabytes of data collected (or gorged on, if you like) during its training. LLMs drive artificial intelligence, and companies spend eye-watering amounts of money on training and running the hardware needed to make these machines work. But the human mind, which is capable of feelings, emotions, and opinions, derives answers to problems elegantly with relatively little information.

Now, I am not a doomsayer at all. I, too, have used and continue to use AI in my daily life. I also have a Copilot subscription, but I was surprised when I watched the recent interview between Microsoft's Satya Nadella and Joanna Stern for *The Wall Street Journal*. Satya Nadella mentioned Copilot+ PCs. These PCs will contain NPUs that will drive AI features on device.

These Copilot+ PCs will have a feature called Recall that will allow users to search over their entire history. It will do this by continually taking screenshots of your desktop and then using generative AI and the NPU to process this data and make it searchable right on your device.

Note: Recall can be restricted from specific websites or apps and always lives on your machine locally. Therefore, the promise is that it can be trusted. Whether this trust will materialize with consumers is a different story altogether.

Artificial intelligence, warts and all, can also significantly benefit humanity. DeepMind AI set out to solve an impossible problem in biology, the protein problem. 3D mapping of a single protein, the building blocks of life, could take years to complete. What DeepMind did was create an AI that could solve the protein problem in a fraction of the time.

They then set it loose on the 200 million proteins that are known to science. Using traditional methods, according to DeepMind CEO Demis Hassabis, this would have taken a billion years to solve. DeepMind took only a year, after which they made their protein database public as a gift to humanity.

While we need to resign ourselves to the fact that AI is here to stay, we as a species have something that no machine will ever have. That is the thirst for, and attainment of, knowledge. The very fact that you are reading this book is a testament to that pursuit of knowledge, driven by an innate desire to learn.

While it is true that AI may assist us, challenge our beliefs, and even change how we as humans live and work, it will never and should never replace our quest for knowledge. As I write these final lines, I am encouraged that our journey of learning is far from over: there will be a .NET 9, and a .NET 10, and so on.

Our future is bright—not because of the tools we use or the solutions we create, but because of the minds that wield them. The human spirit's quest for knowledge is uniquely and profoundly human.

Therefore, dear reader, I thank you for reading this book and allowing me to learn alongside you.