C#Corner

.NET Core

Logging Brilliance in .NET Core

Vinoth Arun

Raj Xavier

Logging Brilliance in Dot Net Core

Vinoth Arun Raj Xavier

Author/Co-Author – Vinoth Arun Raj Xavier
Publisher – C# Corner
Editorial Team – Deepak Tewatia, Baibhav Kumar
Publishing Team – Praveen Kumar
Promotional & Media – Rohit Tomar, Rohit Sharma

About Author

I am Vinoth Arun Raj Xavier, with over 13 years of hands-on experience specializing in Microsoft.NET technology, various JavaScript frameworks, DevOps tools, and other essential programming languages.
My expertise lies in developing Windows, web, and mobile applications. I have a keen interest in microservices, cloud integration, and DevOps practices.
Continuously staying abreast of technological advancements across different technology stacks is something I find personally rewarding.
I focus on writing about current technical trends on social media platforms, sharing knowledge and insights with the community.

Table of Contents:

1

What is Logging?

Overview

In this Chapter, we will explore Logging in software development, highlighting its role in capturing real-time events for issue troubleshooting and performance improvement. It explains ASP.NET Core's logging features, distinguishes logging from monitoring, and outlines popular logging tools. Additionally, it summarizes the six main logging levels in .NET, emphasizing their respective purposes in application development and maintenance.

What is Logging?

Logging in software refers to the process of capturing real-time events along with additional details like infrastructure information and execution time. It plays a crucial role in any software application, particularly when troubleshooting issues. Logs aid in understanding failures, identifying performance bottlenecks, and facilitating problem resolution.
Logs are typically stored in databases, consoles, or files, based on application severity and convenience. While it's possible to record various data in logs, common entries include informational messages and error messages. Informational messages document standard events like method calls, user authentication, or product checkouts, while error messages provide comprehensive data for debugging.
ASP.NET Core simplifies logging by providing a generic logging interface that is consistent across the framework and third-party libraries. This uniformity eases log navigation and problem diagnosis. The framework allows users to configure log verbosity and direct logs to different destinations such as files, consoles, or databases.
In ASP.NET Core, logging providers are responsible for storing logs, and users can configure multiple providers for their applications. The default ASP.NET Core configuration includes logging providers like Console, Debug, EventSource, and EventLog (specific to Windows).

Understanding the Distinction: Logging vs. Monitoring in Software Systems

ASP.NET Core comes with useful features for keeping track of what happens in your applications, known as logging. Logging involves recording detailed information about various events and processes, such as user actions, system events, and performance metrics. This information is typically stored in a log file or database, helping you identify and fix issues, improve performance, and understand how your software is used.
Monitoring, on the other hand, is about watching and measuring your application's performance in real time. It can send alerts or notifications when certain conditions are met, like when the application takes too long to respond or encounters an error. Monitoring helps you catch and fix problems before they impact users, and it provides valuable insights for optimizing performance.
In ASP.NET Core , there are popular tools like Microsoft.Extensions.Logging, NLog, and Serilog that assist with logging. These tools make it easier to manage and analyze the information your application generates, helping you maintain a healthy and efficient software system.
More Details,
In .NET Core, logging is a way to keep track of what's happening in your application. It helps you monitor its behavior, find, and fix issues, and understand how it's performing. The ILogger API is used for efficient and structured logging. Different logging providers allow you to store logs in various places, and there are both built-in and third-party options.
Now, let's talk about the difference between logging and tracing. Logging focuses on recording significant events in your application, providing a kind of summary. On the other hand, tracing goes deeper by giving you a more detailed view of everything happening in your application, offering a complete record of its activities. Think of logs as organized records of important moments, while traces give you a comprehensive look at the entire operation of your application.

The six main logging levels in .NET

- **Critical:** This is for serious issues that could make your app stop working, like running out of memory or space on the disk.
- **Error:** Use this when something goes wrong, like a database error preventing data from being saved. The app can still work for other things despite encountering errors.
- **Warning:** Not as severe as errors, but it indicates a potential problem that might lead to more serious errors. It is a heads-up for administrators.
- **Information:** Gives details about what is happening in the app. Helpful for understanding the steps leading to an error.
- **Debug:** Useful during development for tracking detailed information. It is not typically used in a live/production environment.
- **Trace:** Like Debug but may include sensitive info. Rarely used and not utilized by framework libraries.

2

Serilog Integration for Advanced Logging

Overview

In this chapter, we explore the implementation of an External Logging Source, focusing on Serilog for logging API information efficiently. The chapter begins with a detailed breakdown of the packages required, including Serilog.AspNetCore, with installation instructions provided for both Dot Net CLI and Package Manager. Configuration of Serilog within the API is outlined next, highlighting key settings in the JSON format, such as log levels and log file specifications. Subsequently, the chapter delves into updating the application startup for logging purposes, detailing the necessary steps using Serilog. The setup for logging actions is elucidated, demonstrating the integration of Serilog with the ILogger interface for effective logging into files.

Implement the External Logging Source to Logging Information of API - Serilog

Packages need:

Serilog.AspNetCore

- Dot Net CLI - dotnet add package Serilog.AspNetCore --version 8.0.0
- Package Manager - Install-Package Serilog.AspNetCore -Version 8.0.0

Configuration of Serilog in API

```
 Settings File
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "Serilog": {
    "Using": [ "Serilog.Sinks.File" ],
    "MinimumLevel": {
      "Default": "Information",
      "Override": {
        "Microsoft": "Warning",
        "System": "Warning"
      }
    },
    "WriteTo": [
      {
        "Name": "File",
        "Args": {
          "path": "C:/Log001/log_.log",
          "rollOnFileSizeLimit": true,
          "rollingInterval": "Day"
        }
      }
    ]
  }
}
```

Application Startup updates for Logging

```
using Serilog;
var builder = WebApplication.CreateBuilder(args);
// Add services to the container.
/// Step 1
builder.Host.UseSerilog((context, configuration) =>
{
    configuration.ReadFrom.Configuration(context.Configuration);
}
);
```

```
var app = builder.Build();
/// Step 2
app.UseSerilogRequestLogging();
app.Run();
```

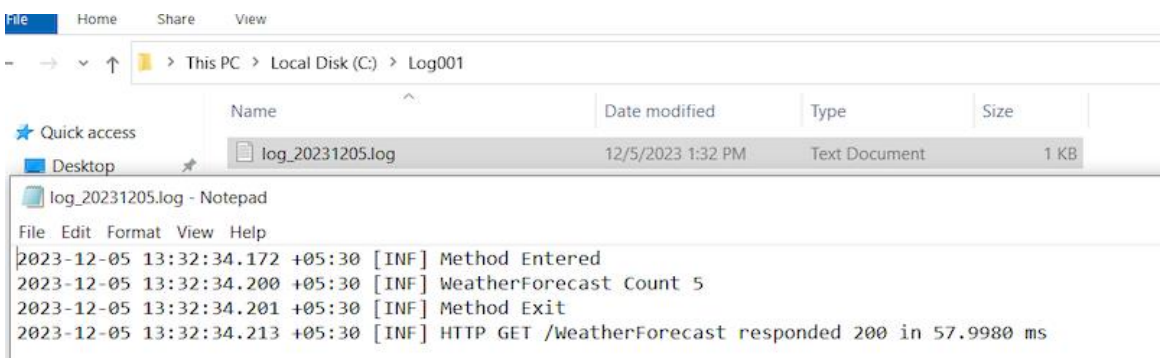
Using the setup to Log actions.

In this setup, we employ the "Assembly - Microsoft.Extensions.Logging.Abstractions, Version=8.0.0.0" for logging actions, and we utilize Serilog to write these logs into files. The ILogger interface is injected into the constructor to facilitate logging functionality.

```
private readonly ILogger<WeatherForecastController> _logger;
public WeatherForecastController(ILogger<WeatherForecastController>
logger)
{
    _logger = logger;
}

[HttpGet(Name = "GetWeatherForecast")]
public IEnumerable<WeatherForecast> Get()
{
    _logger.LogInformation("Method Entered");

    var list = Enumerable.Range(1, 5).Select(index => new
WeatherForecast
    {
        Date = DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
        TemperatureC = Random.Shared.Next(-20, 55),
        Summary = Summaries[Random.Shared.Next(Summaries.Length)]
    })
    .ToArray();

    string message = $"WeatherForecast Count {list.Length}";

    _logger.LogInformation(message);

    _logger.LogInformation("Method Exit");
    return list;
}
```

Output

Now, to test the logging functionality, run the application and inspect the specified path. Refer to the image below for guidance.



3

Configure Serilog for Windows Service Logging

Overview

In this chapter, we implement the setup of external logging for a Windows Service using Serilog. It covers package requirements, Serilog configuration in an API, application startup, and a sample worker class. Readers are encouraged to run the application to observe logging in action.

Implement the External Logging Source to Logging Information of Windows Service - Serilog.

Packages need Serilog.AspNetCore.

```
<PackageReference Include="Serilog.AspNetCore" Version="8.0.0" />
<PackageReference Include="Serilog.Extensions.Hosting" Version="8.0.0"
/>
<PackageReference Include="Serilog.Settings.AppSettings"
Version="2.2.2" />
<PackageReference Include="Serilog.Settings.Configuration"
Version="8.0.0" />
<PackageReference Include="Serilog.Sinks.Console" Version="5.0.1" />
<PackageReference Include="Serilog.Sinks.File" Version="5.0.0" />
```

Confiuration of Serilog in API

Settings File

```
"LogBaseDirectory": "C:\\WindowsServiceLogs\\",
"Serilog": {
  "Using": [ "Serilog.Sinks.Console", "Serilog.Sinks.File" ],
  "MinimumLevel": "Debug",
  "WriteTo": [
    {
      "Name": "Console"
    },
    {
      "Name": "File",
      "Args": {
        "path": "%APP_BASE_DIRECTORY%/Logs/log-.txt",
        "rollingInterval": "Day"
      }
    }
  ]
}
/*
          %APP_BASE_DIRECTORY% - Will be read from App Settings and
update in Startup of Service.
*/
```

Application / Worker Startup

```
using Serilog;
var host = Host.CreateDefaultBuilder(args)
    .UseWindowsService()
    .UseSerilog((context, services, configuration) => configuration
        .ReadFrom.Configuration(context.Configuration)
        .ReadFrom.Services(services))
    .ConfigureServices(services =>
    {
        IConfigurationRoot configuration = new ConfigurationBuilder()
            .SetBasePath(AppDomain.CurrentDomain.BaseDirectory)
            .AddJsonFile("appsettings.json")
            .Build();
```

```
        if (!Directory.Exists(configuration["LogBaseDirectory"]))
        {

Directory.CreateDirectory(configuration["LogBaseDirectory"]);
        }
        Environment.SetEnvironmentVariable("APP_BASE_DIRECTORY",
configuration["LogBaseDirectory"]);

        services.AddHostedService<SimpleWorker>();
    })
    .Build();
await host.RunAsync();
```

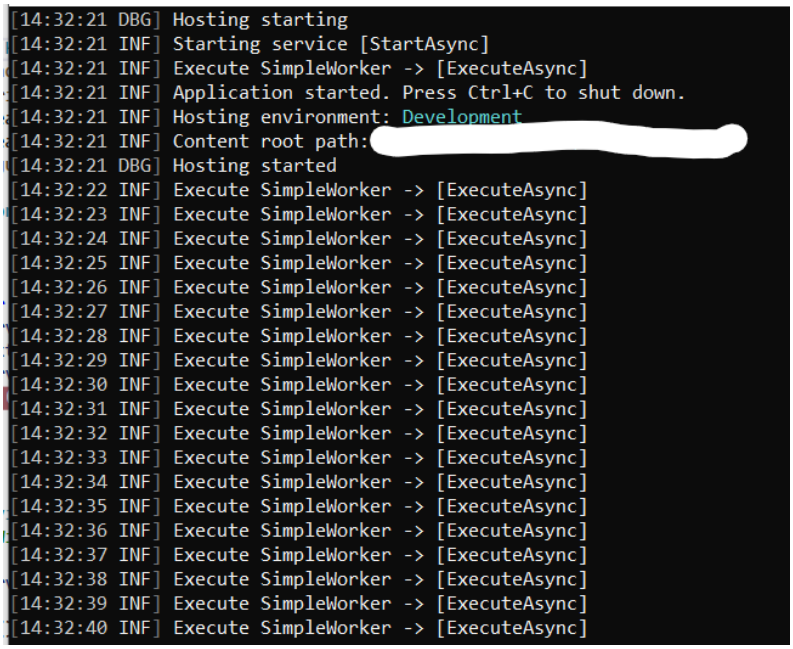
A Worker Class

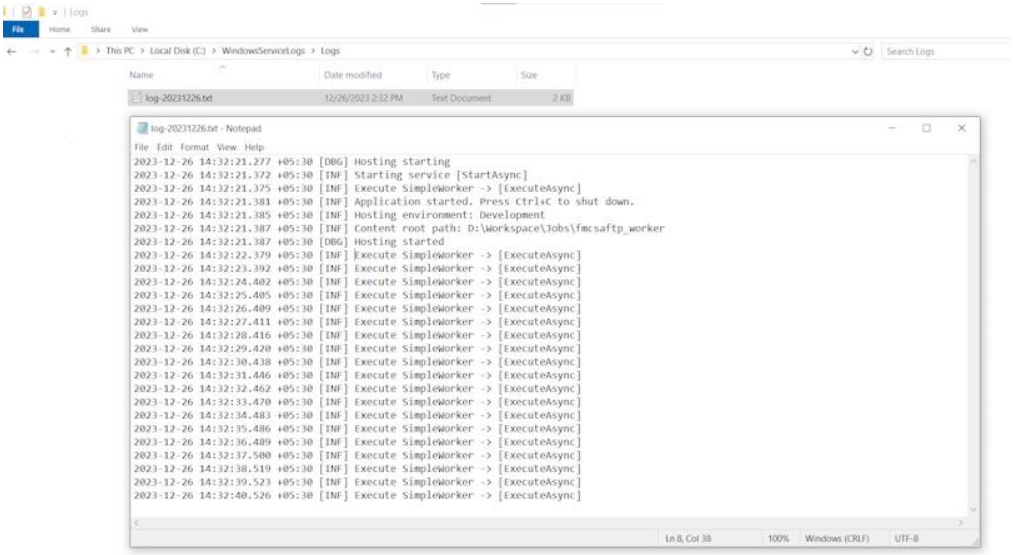
```
public class SimpleWorker : BackgroundService
{
    private readonly ILogger<SimpleWorker> _logger;

    public SimpleWorker(ILogger<SimpleWorker> logger)
    {
        _logger = logger;
    }

    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            _logger.LogInformation("Execute SimpleWorker ->
[ExecuteAsync]");

            await Task.Delay(1000, stoppingToken);
        }
    }


    public override Task StartAsync(CancellationToken
cancellationToken)
    {
        _logger.LogInformation("Starting service [StartAsync]");

        return base.StartAsync(cancellationToken);
    }
    /// <summary>
    /// Executes when the service is performing a graceful shutdown.
    /// </summary>
    /// <param name="cancellationToken"><see
cref="CancellationToken"/></param>
    /// <returns><see cref="Task"/></returns>
```

```
    public override Task StopAsync(CancellationToken
cancellationToken)
    {
        _logger.LogInformation("Stopping service [StopAsync]");

        return base.StopAsync(cancellationToken);
    }
}
```

That's all, we configured all things. Just run the application, See the configured folder check for the log file and console too.

Output:





C#Corner

OUR MISSION

Free Education is Our Basic Need! Our mission is to empower millions of developers worldwide by providing the latest unbiased news, advice, and tools for learning, sharing, and career growth. We're passionate about nurturing the next young generation and help them not only to become great programmers, but also exceptional human beings.

ABOUT US

CSharp Inc, headquartered in Philadelphia, PA, is an online global community of software developers. C# Corner served 29.4 million visitors in year 2022. We publish the latest news and articles on cutting-edge software development topics. Developers share their knowledge and connect via content, forums, and chapters. Thousands of members benefit from our monthly events, webinars, and conferences. All conferences are managed under Global Tech Conferences, a CSharp Inc sister company. We also provide tools for career growth such as career advice, resume writing, training, certifications, books and white-papers, and videos. We also connect developers with their potential employers via our Job board. Visit C# Corner

MORE BOOKS