

Want to kick start your web development in C#? Check out **BLAZOR WEBASSEMBLY COURSE!** 🔥



SEARCH



HOME

BOOK V2

BLAZOR WASM 🔥

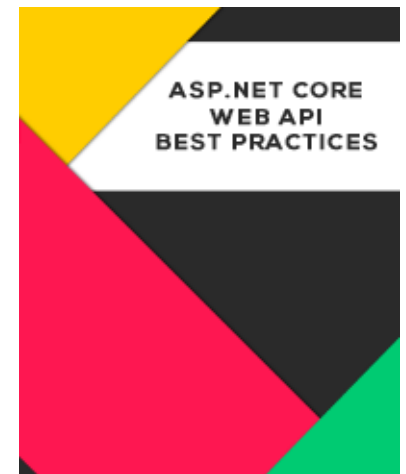
GUIDES ▾

WE ARE HIRING! ▾

ABOUT ▾

Different Log Levels in Serilog

Posted by **Code Maze** | Sep 20, 2023 | 2 🗨️



Made with love by CodeMaze

Join our 20k+ community of experts and learn about our **Top 16 Web API Best Practices.**

Privacy

Want to build **great APIs**? Or become **even better** at it?

Check our **Ultimate ASP.NET Core Web API program**

and learn how to create a full production-ready

ASP.NET Core API using only the latest .NET technologies. Bonus materials (Security book, Docker book, and other bonus files) are included in the Premium package!

In this article, we will discuss the different log levels in Serilog and how to configure them. We will explain the log targets offered by Serilog for writing logs. Additionally, we will cover **how to configure Serilog to differentiate these options according to log levels**.

To download the source code for this article, you can visit our **GitHub repository**.

Setting Up Serilog

Before we can take a look at the different logging levels, we need to configure our project to use Serilog:

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddSerilog(options =>
{
    options.MinimumLevel.Information()
        .WriteTo.Console(LogEventLevel.Information);
});
```

For a more in-depth look at using Serilog, be sure to check out our article **Structured Logging in ASP.NET Core with Serilog**.

Let's start with the log levels.

Different Log Levels in Serilog

Log levels allow us to categorize the seriousness of the logged event. Serilog supports six logging levels:

Support Code Maze on Patreon to get rid of ads and get the best discounts on our products!

 **BECOME A PATRON**

Log Level	Description
Verbose	The lowest log level, enable detailed trace logging mainly for application troubleshooting.

Log Level	Description
Debug	Used for application debugging purposes and to inspect run-time outcomes in development environments.
Information	Used for application monitoring and to track request and response details or specific operation results.
Warning	Used to review potential non-critical, non-friendly operation outcomes.
Error	The most helpful, and yet the most unwanted, log level. Enables detailed error tracking and helps to write error-free applications.
Fatal	The most important log level, used to log critical system operations or outcomes that require urgent attention.

As the log level increases from `Verbose` to `Fatal`, its significance also increases.

Generating Log Messages at Different Log Levels

ASP.NET Core provides the `ILogger<T>` interface for logging scenarios regardless of which logging infrastructure we use. We simply inject the `ILogger<T>` interface into our constructor and straightaway we can begin logging.

With this in mind, let's take a look at generating logs for each log level:

```
public class HomeController : Controller
{
    private readonly ILogger<HomeController> _logger;

    public HomeController(ILogger<HomeController> logger)
```

```
{
    _logger = logger;
}

public IActionResult Index()
{
    _logger.LogTrace("Trace Log Message");
    _logger.LogDebug("Debug Log Message");
    _logger.LogInformation("Information Log Message");
    _logger.LogWarning("Warning Log Message");
    _logger.LogError("Error Log Message");
    _logger.LogCritical("Critical Log Message");

    return View();
}
```

As we mentioned earlier, in the `HomeController` constructor, we inject the `ILogger<HomeController>` interface, which we use to initialize our `_logger` instance. Then, in our `Index()` method, using our `_logger`, we log a message at each of the supported log levels.

Configuring Minimum Log Levels in Serilog

Logging is an essential part of our applications. Through the data it provides, we can detect failures, problems, and performance issues in our code. However, logging everything may not always be a desired scenario. Due to this, we may need to configure our application to log only specific log levels.

Additionally, we may need to generate environment-specific logs. For example, **during development, we may want to see all logging levels between `Debug` and `Fatal`**. On the other hand, **in our production environment, we probably want to restrict this to only `Warning`, `Error`, and `Fatal` levels, to prevent consuming too many resources through logging.**

Here is where Serilog's `MinimumLevel` configuration comes to our rescue. **If we configure this setting with a log level, then Serilog only generates logs for that level and higher.** For example, if we set the `MinimumLevel` as `Warning`, then Serilog records only `Warning`, `Error` and `Fatal` level logs. If we don't specify a `MinimumLevel` setting, Serilog uses the `Information` log level as the default. We can configure this setting in either the `appSettings.json` file or via the fluent API in `Program.cs`.

Let's take a look at configuring `MinimumLevel` in the `appSettings.json` file:

```
"Serilog": {  
  "MinimumLevel": {  
    "Default": "Information",
```

```
"Override": {  
    "Microsoft": "Warning",  
    "System": "Warning"  
}  
}  
}
```

Here, we instruct Serilog to generate application logs for levels equal to or higher than the `Information` log level. Additionally, we override the logging of Microsoft and System messages, ensuring they are recorded solely when they are of `Warning` level or higher.

Now let's perform the same configuration via the fluent API in `Program.cs`:

Wanna join Code Maze Team, help us produce more awesome .NET/C# content and **get paid? >> JOIN US! <<**

```
builder.Services.AddSerilog(options =>  
{
```

```
options.MinimumLevel.Information()  
    .MinimumLevel.Override("Microsoft", LogEventLevel.Warning)  
    .MinimumLevel.Override("System", LogEventLevel.Warning)  
});
```

Where to Log?

We can configure Serilog to write logs to different targets or sinks. **A sink, in the context of logging, is the destination for our log messages.** Here are some of the most common sinks:

- The application console
- The file system
- Relational databases, like MS SQL Server
- Non-relational databases, like Elasticsearch or MongoDB

A complete list of available sinks can be found [here](#). For a more in-depth look at sinks check out our article [Structured Logging in ASP.NET Core with Serilog](#).

We can configure Serilog sinks in `appSettings.json` file or via fluent API in `Program.cs` file.

Configuring Sinks in appSettings.json

Let's configure a console and a file sink via `appSettings.json`:

```
"Serilog": {  
  "Using": [ "Serilog.Sinks.File", "Serilog.Sinks.Console" ],  
  "WriteTo": [  
    {  
      "Name": "Console",  
      "Args": {
```



```

        "restrictedToMinimumLevel": "Information",
        "outputTemplate": "[{Timestamp:HH:mm:ss} {Level:u3}] {Mess
    }
},
{
    "Name": "File",
    "Args": {
        "path": "logs/log-.txt",
        "rollOnFileSizeLimit": true,
        "rollingInterval": "Day",
        "fileSizeLimitBytes": "1000000",
        "outputTemplate": "[{Timestamp:HH:mm:ss} {Level:u3}] {Mess
        "restrictedToMinimumLevel": "Warning"
    }
}
]
}

```

Firstly, we specify the sinks, via `Using` option. Then, we configure each of them through the `WriteTo` option. This takes an array of sinks with their associated configuration settings.

Configuring Sinks via Fluent API

Now let's see how we can configure our logging sinks using the fluent API in

`Program.cs`:

```

builder.Services.AddSerilog(options =>
{
    options.MinimumLevel.Information()
        .MinimumLevel.Override("Microsoft", LogEventLevel.Warning)
        .MinimumLevel.Override("System", LogEventLevel.Warning)
        .WriteTo.Console(restrictedToMinimumLevel: LogEventLevel.Information,
            outputTemplate: "[{Timestamp:HH:mm:ss} {Level:u3}] {Message}")
        .WriteTo.File("logs/log-.txt",

```

```
rollOnFileSizeLimit: true,  
rollingInterval: RollingInterval.Day,  
fileSizeLimitBytes: 1000000,  
outputTemplate: "[{Timestamp:HH:mm:ss} {Level:u3}]"  
restrictedToMinimumLevel: LogEventLevel.Warning);  
});
```

Differentiating Log Sinks Based on Log Level

Based on application requirements and environment, we may need to configure Serilog to write specific log levels to specific sinks. For example, we



but logs with the minimum log level `Warning` to a file. Under these circumstances, we use the `restrictedToMinimumLevel` setting. **It allows Serilog to differentiate log destinations based on the minimum log level.**

We can configure the `restrictedToMinimumLevel` setting through `appSettings.json`:

Wanna join Code Maze Team, help us produce more awesome .NET/C# content and **get paid? >> JOIN US! <<**

```
"Serilog": {  
  "Using": [ "Serilog.Sinks.File", "Serilog.Sinks.Console" ],  
  "WriteTo": [  
    {  
      "Name": "Console",  
      "Args": {  
        "restrictedToMinimumLevel": "Information"  
      }  
    },  
    {  
      "Name": "File",  
      "Args": {  
        "path": "logs/log-.txt",  
        "restrictedToMinimumLevel": "Warning"  
      }  
    }  
  ]  
}
```

In like fashion, we can also configure `restrictedToMinimumLevel` through the fluent API:

```
builder.Services.AddSerilog(options =>
{
    options.WriteTo.Console(restrictedToMinimumLevel: LogEventLeve
        .WriteTo.File("logs/log-.txt",
            restrictedToMinimumLevel: LogEventLevel.Warning
        ));
});
```

Conclusion

In this article, we have explored different log levels in Serilog and how to use them in our application. We delved into configuring the Serilog environment specifically based on the `MinimumLevel` parameter. Subsequently, we talked about Serilog sinks and how to configure these sinks through either the `appSettings.json` file or through the fluent API. Finally, we inspected the `restrictedToMinimumLevel` parameter and how it allows us to differentiate log sinks based on log levels.

Liked it? Take a second to support Code Maze on Patreon and get the ad free reading experience!

 **BECOME A PATRON**



Want to build **great APIs**? Or become **even better** at it?

Check our **Ultimate ASP.NET Core Web API program**

and learn how to create a full production-ready

ASP.NET Core API using only the latest .NET

technologies. Bonus materials (Security book, Docker

book, and other bonus files) are included in the

Premium package!

SHARE:   

 [Subscribe](#) ▼

[Login](#)



Join the discussion

B *I* U    “ ” </>  {} [+]



2 COMMENTS



Newest ▼



Alex

 3 months ago

How to map serilog settings to Microsoft logger settings?



0




Reply

[Privacy](#)



Osman Sokuoğlu

 Reply to [Alex](#)

 3 months ago

In the "Setting up Serilog" section when you add serilog to your project no need to map any settings to microsoft logger. ASP.NET Core automatically reads the Serilog configuration and when you use Microsoft's ILogger in your project it applies Serilog configuration.

+ 0 -  Reply



© Copyright code-maze.com 2016 - 2024