

What is Serilog?

[Serilog is a third-party logging library](#) that plugs into the default ILogger of our application with its own implementations. It enables the developers to log the events into various destinations like console, file, database, and more. Now, if you are already using a database in your ASP.NET Core Application, logging events to a database can be a good option. Serilog supports structured logging, which allows more details and information about the event to be logged. With structured logging in place, you could use these logs to debug in a very logical way.

Setting up the ASP.NET Core 3.1 Project

For this demonstration, let's implement Serilog on an ASP.NET Core 3.1 WebApplication (Razor Pages). Since our focus is on logging and understanding various related concepts, we will keep the project setup simple and straight-forward. I will be using [Visual Studio 2019 Community](#) as my IDE.

Logging with the Default Logger

As I had mentioned earlier, ASP.NET Core applications ship with a default built-in logging system which includes some basic logging functions. To understand logging, let's see how the basic logger works. Once you have created your WebApplication solution, navigate to Pages / Index.cshtml / Index.cshtml.cs. You can see the contractor injection of the ILogger interface. This is the default logger from Microsoft.

In the OnGet method of the IndexModel, let's add a way to demonstrate logging and also use the try-catch block . Here I will throw a Dummy Exception so that we can understand logging better. Also note that we will not be changing anything on the class further in this demonstration.

```
public void OnGet()
{
    _logger.LogInformation("Requested the Index Page");
    int count;
    try
    {
        for(count = 0;count<=5;count++)
        {
            if(count==3)
            {
```

```
throw new Exception("RandomException");
}
}
}
catch (Exception ex)
{
    _logger.LogError(ex, "Exception Caught");
}
}
}
```

The OnGet method is fired every time you request for the Index Page (Home Page). So, as the code suggests, I am logging a message that says "Requested the Index Page" every time you request for this page. After that it runs a loop 5 times, and if the iteration count is 3, it throws a dummy exception "RandomException" which in turn gets caught in the catch block. This is logged as an error. This way, we have a function that mimics a practical production level function.

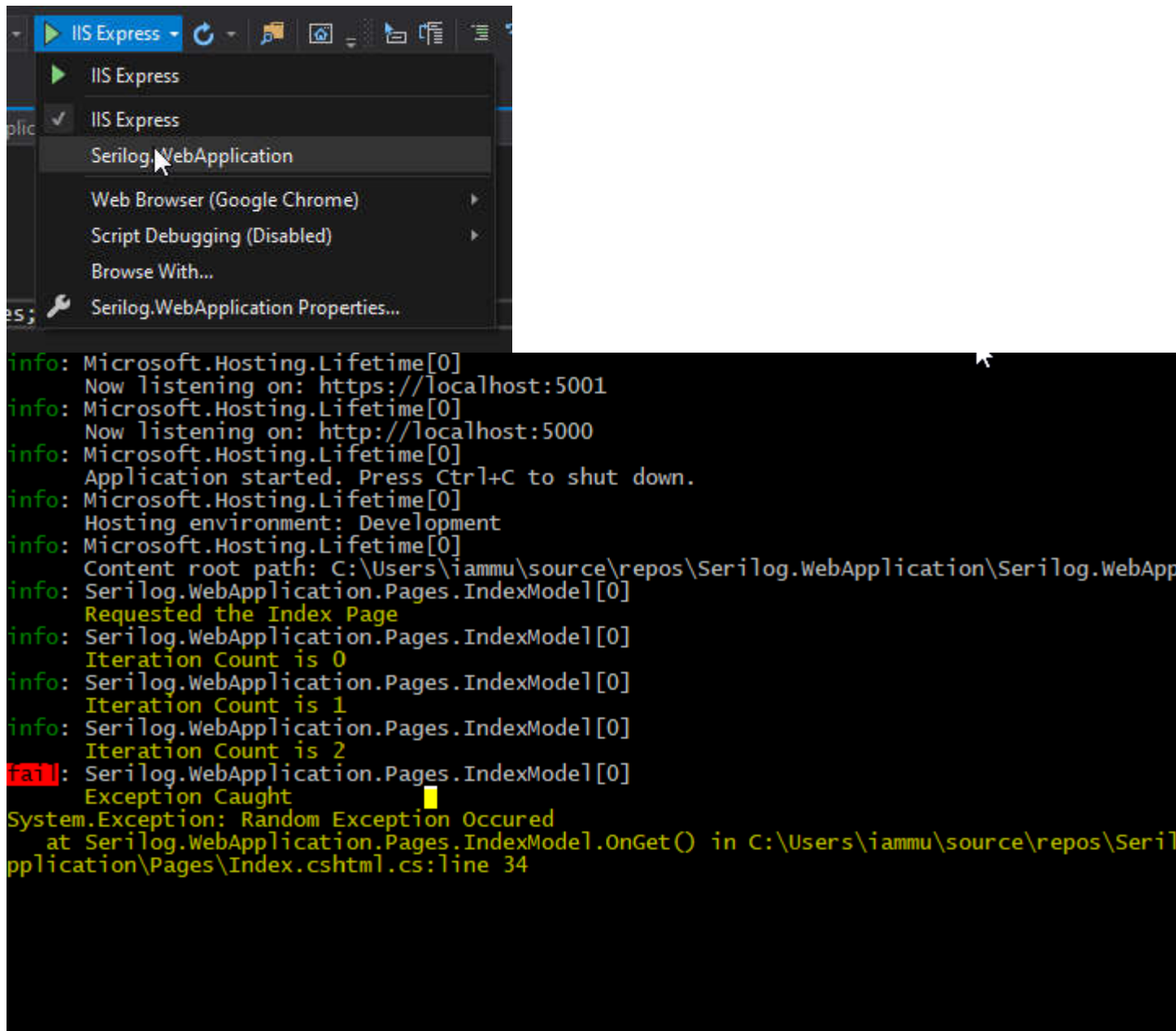
Before testing the logging, let's switch to the Kestrel web server from IIS Express on Visual Studio.

What is Kestrel Webserver?

Kestrel is an open-source web server that ships by default with ASP.NET Core applications. We specifically need the Kestrel server for this demonstration because it opens up a console that contains all the logged events.

How to switch to Kestrel Webserver?

By default, you might have IIS Express as the selected server. You can switch to Kestrel Web Server by hitting the dropdown icon beside the IIS Express and choose the option with your application name. In our case, it is Serilog.WebApplication. I will choose it and run the application using Control + F5 (I am starting the application without debugging to save some time here).



When you run your application, a console opens up along with your web application. In this console, you see certain logs from the application. In the end, we see our custom Log Messages and Exceptions (I have highlighted our concerned messages with yellow). Try to refresh the page on your web browser, you will see another set of similar messages on the console. This is the default logging provided with your application.

Log Levels

I also wanted you to know about the various Logging Levels. This is the fundamental concept of logging. When we wrote '`_logger.LogInformation("Requested the Index Page");`', we mentioned

to the application that this is a log with the log-level set to Information. Log levels make sense because it allows you to define the type of log. Is it a critical log ? just a debug message? a warning message?

There are 7 log-levels included :

- Trace – Detailed messages with sensitive app data.
- Debug – Useful for the development environment.
- Information – General messages, like the way we mentioned earlier.
- Warning – For unexpected events.
- Error – For exceptions and errors.
- Critical – For failures that may need immediate attention.

Note that Serilog may or may not have the same names for each level, but you get the idea, right? [You can read more about log levels here.](#)

Default Log Settings

The default settings for our logger is mentioned in appsettings.json. These settings allows you to define on what level of logs you need from a particular component. For example, any log messages that is generated by the Application (Microsoft) with levels Warning and above is logged to the console. This is the basic idea of log settings.

```
"Logging": {  
  "LogLevel": {  
    "Default": "Information",  
    "Microsoft": "Warning",  
    "Microsoft.Hosting.Lifetime": "Information"  
  }  
}
```

With that out the way, let's start the actual implementation of Serilog in our ASP.NET Core application.

Serilog Enrichers

To enable Structured Logging and to unleash the full potential of Serilog, we use enrichers. These enrichers give you additional details like Machine Name, ProcessId, Thread Id when the log event had occurred for better diagnostics. It makes a developer's life quite simpler. We will use the enrichers later in this guide.

Serilog Sinks

Serilog Sinks in simpler words relate to destinations for logging the data. In the packages that we are going to install to our ASP.NET Core application, Sinks for Console and File are included out of the box. That means we can write logs to Console and File System without adding any extra packages. Serilog supports various other destinations like MSSQL, SQLite, SEQ and more.

Implementing Serilog in ASP.NET Core 3.1

Let's start implementing Serilog in our ASP.NET Core 3.1 Application and make it the default logger application-wide. Here is a quick step-by-step guide on How to use Serilog in ASP.NET Core Applications.



Installing the Required Packages

For now, these are the packages that you require. Install them via the NuGet Package Manager or Console.

Install-Package Serilog.AspNetCore

Install-Package Serilog.Settings.Configuration

Install-Package Serilog.Enrichers.Environment

Install-Package Serilog.Enrichers.Process

Install-Package Serilog.Enrichers.Thread

Package #1 contains the core components of Serilog. This is an ASP.NET Core version of the package which comes along with additional features for our application.

Package #2 allows Serilog to read the settings from our configuration file, ie appsettings.json.

Package #3,4,5 are the enrichers that get details of the environment, process, thread, etc.

Now that we have installed all the necessary Serilog packages, let's go ahead and configure Serilog.

Configuring Serilog in ASP.NET Core Applications

Our intention is to use Serilog instead of the default logger. For this, we will need to configure Serilog at the entry point of our ASP.NET Core Application, ie, the Program.cs file. Navigate to Program.cs and make the following changes.

```
public static void Main(string[] args)
{
    //Read Configuration from appSettings
    var config = new ConfigurationBuilder()
        .AddJsonFile("appsettings.json")
        .Build();
    //Initialize Logger
    Log.Logger = new LoggerConfiguration()
        .ReadFrom.Configuration(config)
        .CreateLogger();
    try
    {
        Log.Information("Application Starting.");
        CreateHostBuilder(args).Build().Run();
    }
    catch (Exception ex)
    {
        Log.Fatal(ex, "The Application failed to start.");
    }
    finally
    {
        Log.CloseAndFlush();
    }
}
```

```
}
```

Explanation.

Line #4,5,6 reads a config file (appsettings.json) and get the Configuration object for later use.

Line #9,10,11 Initiliazlies the Serilog using the settings from appsettings.json

Line #23 allows the logger to log any pending messages while the application closes down.

public static IHostBuilder CreateHostBuilder(string[] args) =>

Host.CreateDefaultBuilder(args)

.UseSerilog() //Uses Serilog instead of default .NET Logger

.ConfigureWebHostDefaults(webBuilder =>

```
{
```

webBuilder.UseStartup<Startup>());

```
});
```

Line #3 makes the application use Serilog instead of the default Logger.

NOTE – It is important to note that, in this tutorial I am showing you a cleaner way to implement Serilog. There are possibilities to define the Serilog Configuration in Code (C#). But the issue is, you can not modify these settings at runtime. Hence it is a better practice to define these settings in appsettings.json.

Now that our Application is configured to support Serilog as the default logger, let's define Serilog Settings in our appsettings.json.

Setting up Serilog

Navigate to appsettings.json and remove the default logging settings and replace it with the following.

```
{
```

"AllowedHosts": "*",

"Serilog":

```
{
```

"Using": [],

"MinimumLevel": {

"Default": "Information",

"Override":

```
{
```

```

"Microsoft": "Warning",
"System": "Warning"
},
"WriteTo": [
{
"Name": "Console"
},
{
"Name": "File",
"Args": {
"path": "D:\\Logs\\log.txt",
"outputTemplate": "{Timestamp} {Message} {NewLine:1} {Exception:1}"
}
},
],
"Enrich": [
"FromLogContext",
"WithMachineName",
"WithProcessId",
"WithThreadId"
],
"Properties": {
"ApplicationName": "Serilog.WebApplication"
}
}
}

```

Explanation.

This is the Settings for Serilog defined in appsettings.json. As I had mentioned earlier, from now on, we will be only changing the settings here and won't touch C# code.

Line #3 marks the beginning of the Serilog Settings.

Line #6 to #13 defined the minimum level of logging for various and default components. You

can see that by default, we log all the levels above Information Log Level. But for specific components like Microsoft, we need to log only for the Warning and above levels, so that we don't have a trillion lines of log in our sinks.

Line #14 to 25 marks the Serilog Sink Settings. For now, we have written the settings for File and Console Sinks only. We will extend it further in this guide.

Line #22 is where you can define the template of the log output.

Line #26 to 31 defines the enrichers for Serilog to provide more details.

Line #32 to 34 is where we can define custom properties that will appear in our structured log data.

Hope this part is clear. Let's move forward and run the application. Make sure Kestrel is your default webserver.

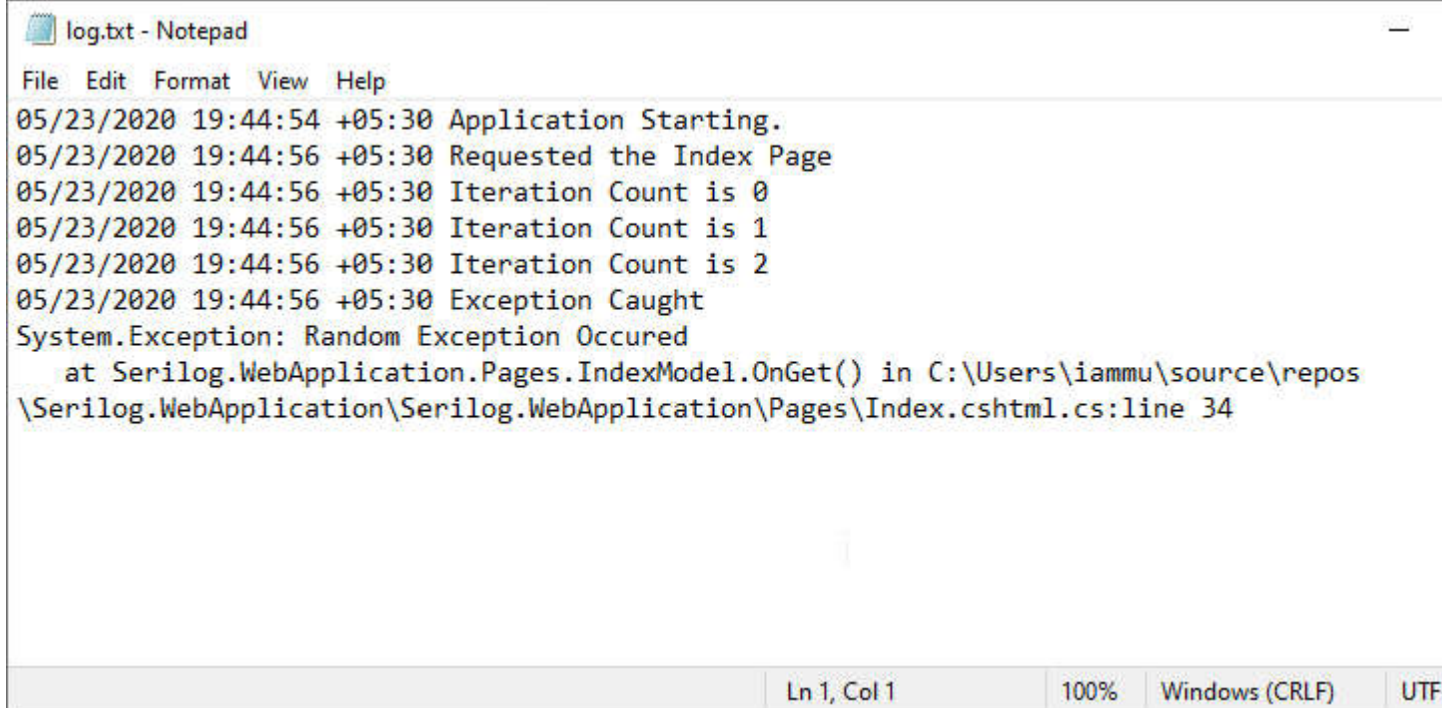
Logging to Console with Serilog.

We do not need any extra changes to log to the console. You can see that our log is now much cleaner and to the point.

```
[19:44:54 INF] Application Starting.
[19:44:56 INF] Requested the Index Page
[19:44:56 INF] Iteration Count is 0
[19:44:56 INF] Iteration Count is 1
[19:44:56 INF] Iteration Count is 2
[19:44:56 ERR] Exception Caught
System.Exception: Random Exception Occured
   at Serilog.WebApplication.Pages.IndexModel.OnGet() in C:\Users\iammu\source\repos\Serilog.WebApplication\Pages\Index.cshtml.cs:line 34
```

Logging to File with Serilog.

Now, let's check the folder that we had defined earlier. We can see a new log.txt file created by Serilog.



```
log.txt - Notepad
File Edit Format View Help
05/23/2020 19:44:54 +05:30 Application Starting.
05/23/2020 19:44:56 +05:30 Requested the Index Page
05/23/2020 19:44:56 +05:30 Iteration Count is 0
05/23/2020 19:44:56 +05:30 Iteration Count is 1
05/23/2020 19:44:56 +05:30 Iteration Count is 2
05/23/2020 19:44:56 +05:30 Exception Caught
System.Exception: Random Exception Occured
    at Serilog.WebApplication.Pages.IndexModel.OnGet() in C:\Users\iammu\source\repos\Serilog.WebApplication\Serilog.WebApplication\Pages\Index.cshtml.cs:line 34
Ln 1, Col 1 100% Windows (CRLF) UTF
```

Ok, We have talked so much about Structured Logging and Enrichers. But where are they? Console and File(text files) Sinks don't support Structured Logging. Let's move to the next schema to achieve structured logging.

Structured Logging with Serilog.

To enable structured logging with the File Sink, we need to add a JSON formatter as a Parameter to the Settings. Let's add new Sink to our appSettings.json/Serilog. Add the below code as the sink.

```
{
  "Name": "File",
  "Args": {
    "path": "D:\\Logs\\structuredLog.json",
    "formatter": "Serilog.Formatting.Json.JsonFormatter, Serilog"
  }
}
```

Line #2, this is File Sink.

Line #4 defines the path of the JSON File.

Line #5 seats the JSON Formatter of Serilog to enable structured logging.

Now, just run the application again. Navigate to the defined path, you will see a new JSON file.

Open this file with a Code Editor that can format JSONs, so that it is easier to read through the data. I used Visual Studio Code to open the log file. Here is a screenshot of the log.

```
91  {
92      "Timestamp": "2020-05-23T19:44:56.6799405+05:30",
93      "Level": "Error",
94      "MessageTemplate": "Exception Caught",
95      "Exception": "System.Exception: Random Exception Occured\r\n    at Serilog.WebAppl
96      "Properties": {
97          "SourceContext": "Serilog.WebApplication.Pages.IndexModel",
98          "ActionId": "6c6ee332-f542-42d8-8021-5b6480b05056",
99          "ActionName": "/Index",
100         "RequestId": "0HLVV4ELJ1A2S:00000001",
101         "RequestPath": "/",
102         "SpanId": "|10c3e39f-42d6d6fb994bcd61.",
103         "TraceId": "10c3e39f-42d6d6fb994bcd61",
104         "ParentId": "",
105         "MachineName": "DESKTOP-QCM5AL0",
106         "ProcessId": 21328,
107         "ThreadId": 8,
108         "ApplicationName": "Serilog.WebApplication"
109     }
110 }
```

This is the Structured Log of a single log Event, the Error event. You can see the amount of data given to us by Serilog. Also note that we have the Machine Name, ProcessId, and many details that can help debug the application in the longer run. Our Custom Property, ApplicationName also appears in our Log File. Pretty cool, yeah?

Logging to Database with Serilog.

Now, this is what you would probably want to use for applications at production environment.

It makes much more sense to log into a relational database, so that , at a later point of time, you could use some queries to intelligently fetch the log details by AppliationName,LogLevel, etc.

In this tutorial, I will show you the way to log to Microsoft SQL Server Database. For this, we need to install an additional package (a new Sink!). It is possible to log to multiple databases as well (you will have to install the specific Serilog Sink package).

Install-Package Serilog.Sinks.MSSqlServer

```
{  
  "Name": "MSSqlServer",  
  "Args": {  
    "connectionString": "<your connection string / named connection Here>",  
    "sinkOptionsSection": {  
      "tableName": "Logs",  
      "schemaName": "EventLogging",  
      "autoCreateSqlTable": true  
    },  
    "restrictedToMinimumLevel": "Warning"  
  }  
}
```

Line #2 defined the sink as MSSQLServer Sink.

Line #4 is where you would want to put in the connection string / named connection string for the logs to be inserted.

Line #6 the table name.

Line #7 the name of the schema.

Line #8 is a cool feature where Serilog creates the table for you if it does not exist.

Line #10 restricts the level of minimum log level. We do not want to log everything on to the database during production. Just the Errors and Fatal Logs are Enough.

Let me fire up the application. It may take a few more seconds than usual because Serilog is setting up your database in the background. After the application executes. Open up SQL Management Studio and check the database mentioned in your connection string. You will find a new Table "Eventlogging.Logs". This is the one created by Serilog. Run a Select * Command on this table.

```

/***** Script for SelectTopNRows command from SSMS *****/
SELECT TOP (1000) [Id]
      ,[Message]
      ,[MessageTemplate]
      ,[Level]
      ,[TimeStamp]
      ,[Exception]
      ,[Properties]
FROM [DevelopmentDatabase].[EventLogging].[Logs]

```

100 %

Results Messages

	Id	Message	MessageTemplate	Level	TimeStamp	Exception	
1	1	Exception Caught	Exception Caught	Error	2020-05-23 20:32:25.407	System.Exception: Random Exception Occured	...

Note that our MSSqlServer sink only logs the events which are high priority, ie, Errors / Exceptions / Fatafs. This can be set for any sink as well depending on your requirement. Great! Now we have learnt a clean way to implement Serilog / Logging in your ASP.NET Core application.

Summary

In this article, we have gone through with basics of logging, the behavior of default .NET logger, various concepts of logging, Serilog library, and it's implementation, and different Serilog Sinks. What is your favorite Logging Framework? Is it not Serilog ? Well, mine is. ***In the next section you can find the link to the source code that I demonstrated in this article.*** Leave behind your queries , suggestions in the comment section below. Also, if you think that you learnt something new from this article, do not forget to share this within your developer community. Happy Coding!