**Create Your Own Logging Provider to Log to Text Files in .NET Core**

23rd July 2020

<https://www.roundthecode.com/dotnet/create-your-own-logging-provider-to-log-to-text-files-in-net-core>

One of the things that .NET Core supports is a logging library. This library works with a number of built-in and third-party logging providers. And, you can focus on particular areas of your application that you wish to provide logging to.

We are going to explore some of the basics of logging in .NET Core and then write our own custom logging provider. The custom logging provider will enable us to output logging to text files.

**Configuring Logging**

When you create an ASP.NET Core API project, an appsettings.json file will be created similar to this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | {    "Logging": {      "LogLevel": {        "Default": "Information",        "Microsoft": "Warning",        "Microsoft.Hosting.Lifetime": "Information"      }    },    "AllowedHosts": "\*"  } |

It's worth noting the different log levels available before we continue.

LogLevel in Microsoft.Extensions.Logging is an enum that provides all the different log levels. Here are the values and indexes available to you:

* Trace - 0
* Debug - 1
* Information - 2
* Warning - 3
* Error - 4
* Critical - 5
* None - 6

We are going to ignore "None" in this instance. So the way it works in appsettings.json is that when you specify a log level, you provide a JSON property. The property key represents the namespace and the property value represents the minimum level of logging. A namespace which is deeper will take priority.

|  |  |
| --- | --- |
| 1  2  3 | "Default": "Information",  "Microsoft": "Warning",  "Microsoft.Hosting.Lifetime": "Information" |

So in the example from above, logging code that sits in the Microsoft namespace will log any events that are higher or equal to Warning. So, warning, error and critical events will be logged. However, any code that sits in the Microsoft.Hosting.Lifetime namespace will also have information logs captured, in addition to warning, error and critical events.

Now that we have that understanding, we can use one of the built in logging providers to customise how we want to output logging information. We are going to use Console as it's one of the easiest to demonstrate.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | {    "Logging": {      ...,      "Console": {        "LogLevel": {          "Default": "Error",          "Microsoft": "Error"        }      }    },    ...  } |

This means that when we start our ASP.NET Core API application in Visual Studio, it's only going to output errors to the console. Assuming we don't have any errors, it's not going to output any events.

**Adding Log Events**

We are now going to add log events into an API controller. The first thing we need to do is to specify the ILogger as a parameter, passing in the type as it's generic method.

The ILogger gets injected through dependency injection and will allow us to log events. The generic type will tell the logger which namespace it's part of.

For example, with a class named ArticleHitController, we would inject ILogger<ArticleHitController> as a parameter.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | // ArticleHitController.cs  [ApiController]  [Route("api/article-hit")]  public class ArticleHitController : Controller  {      ...      protected readonly ILogger<ArticleHitController> \_logger;        public ArticleHitController(..., [NotNull] ILogger<ArticleHitController> logger)      {          ...          \_logger = logger;      }        ...  } |

Then it's a case of using the ILogger instance to log the relevant events. In the example below, we are logging events for our API endpoints. We can use the LogInformation method to provide information logs. And you can use the LogTrace method to provide trace logs.

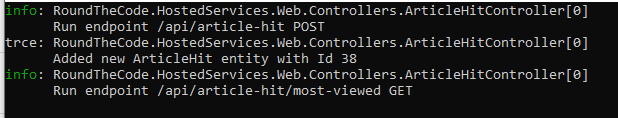
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | // ArticleHitController.cs  [ApiController]  [Route("api/article-hit")]  public class ArticleHitController : Controller  {      ...        [HttpPost]      public async Task<IActionResult> CreateAsync(ArticleHit entity)      {          \_logger.LogInformation("Run endpoint {endpoint} {verb}", "/api/article-hit", "POST");            await \_hostedServicesDbContext.Set<ArticleHit>().AddAsync(entity);          await \_hostedServicesDbContext.SaveChangesAsync();            \_logger.LogTrace("Added new ArticleHit entity with Id {id}", entity.Id);            return Ok(entity);      }        [HttpGet("most-viewed")]      public IActionResult GetMostViewed()      {          \_logger.LogInformation("Run endpoint {endpoint} {verb}", "/api/article-hit/most-viewed", "GET");            return Ok(\_mostViewedArticleService.MostViewedArticles);      }  } |

As you can see, we have set up two endpoints with a mixture of logging at trace and information levels.

Now, the assembly for this controller sits in RoundTheCode.HostedServices.Web.Controllers. In-order to output the log to the console, we need to make a change to our appsettings.json file.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | {    "Logging": {      ...,      "Console": {        "LogLevel": {          "Default": "Error",          "Microsoft": "Error",          "RoundTheCode.HostedServices.Web.Controllers": "Trace"        }      },    },    ...  } |

We are now outputting logs in the console where the namespace is RoundTheCode.HostedServices.Web.Controllers. As a result, running our API and calling an API endpoint will output logs to the console.



**Create a Custom Logging Provider**

We have a scenario where we want to log the output of files to a text file. Now, there are plenty of third-party logging providers out there that will do this for you. However, there is a chance that they don't support a feature that you want and you may have to compromise.

But it doesn't need to be that way. Setting up a custom logging provider is very simple in .NET Core and we will run you through the steps on how to do it.

**Options**

We want to use the appsettings.json file to store our configurations. When outputting log files to text files, we will need to know the folder path and the format of the log file. So we can make a class that can store this information.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | // RoundTheCodeFileLoggerOptions.cs  public class RoundTheCodeFileLoggerOptions  {      public virtual string FilePath { get; set; }        public virtual string FolderPath { get; set; }  } |

**Logger Provider**

Next, we want to create the logger provider file. The logger provider file will inject our options that we will eventually get from our appsettings.json file.

The logger provider file needs to inherit ILoggerProvider. One of the methods we need to declare is CreateLogger. This will create a new instance of our Logger class that we will create next.

In-addition, we specify a parameter which will be an instance of the logger provider. This is so we can retrieve our options like the folder path and file path and can successfully output log entries to text files.

Finally, we need to include the ProviderAlias attribute. The ProviderAlias attribute helps us identity the name of the logger provider in appsettings.json.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | // RoundTheCodeFileLoggerProvider.cs  [ProviderAlias("RoundTheCodeFile")]  public class RoundTheCodeFileLoggerProvider : ILoggerProvider  {      public readonly RoundTheCodeFileLoggerOptions Options;        public RoundTheCodeFileLoggerProvider(IOptions<RoundTheCodeFileLoggerOptions> \_options)      {          Options = \_options.Value;            if (!Directory.Exists(Options.FolderPath))          {              Directory.CreateDirectory(Options.FolderPath);          }      }        public ILogger CreateLogger(string categoryName)      {          return new RoundTheCodeFileLogger(this);      }        public void Dispose()      {      }  } |

**Logger**

Next, we want to create the logger file. This inherits the ILogger interface and allows us to do what we want with the log event. In the constructor, we have to pass in the logger provider as a parameter. This is called inside the CreateLogger method in the logger provider class.

Then we go ahead and put the folder path and file path together so we can create the full file path. This will help us save the log file in the correct location.

Thereafter, we create the log record ouput as a string and then write it to the log file.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | // RoundTheCodeFileLogger.cs  public class RoundTheCodeFileLogger : ILogger      {          protected readonly RoundTheCodeFileLoggerProvider \_roundTheCodeLoggerFileProvider;            public RoundTheCodeFileLogger([NotNull] RoundTheCodeFileLoggerProvider roundTheCodeLoggerFileProvider)          {              \_roundTheCodeLoggerFileProvider = roundTheCodeLoggerFileProvider;          }            public IDisposable BeginScope<TState>(TState state)          {              return null;          }            public bool IsEnabled(LogLevel logLevel)          {              return logLevel != LogLevel.None;          }            public void Log<TState>(LogLevel logLevel, EventId eventId, TState state, Exception exception, Func<TState, Exception, string> formatter)          {              if (!IsEnabled(logLevel))              {                  return;              }                var fullFilePath = \_roundTheCodeLoggerFileProvider.Options.FolderPath + "/" + \_roundTheCodeLoggerFileProvider.Options.FilePath.Replace("{date}", DateTimeOffset.UtcNow.ToString("yyyyMMdd"));              var logRecord = string.Format("{0} [{1}] {2} {3}", "[" + DateTimeOffset.UtcNow.ToString("yyyy-MM-dd HH:mm:ss+00:00") + "]", logLevel.ToString(), formatter(state, exception), exception != null ? exception.StackTrace : "");                using (var streamWriter = new StreamWriter(fullFilePath, true))              {                  streamWriter.WriteLine(logRecord);              }          }      } |

**Add Options to appsettings.json**

Next, we need to specify our options into appsettings.json. Now if you remember, we provided a ProviderAlias attribute in our logger provider class and called it RoundTheCodeFile. We now need to specify this in the Logging property.

We need to specify the log levels so we know which events we want logged to text files.

In addition, we need to specify the options. We haven't configured the options to be read in our application yet, but it will need to be done for it to work properly.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | {    "Logging": {      ...,      "RoundTheCodeFile": {        "Options": {          "FolderPath": "C:\\Users\\myusername\\source\\repos\\RoundTheCode.HostedServices\\RoundTheCode.HostedServices.Web\\logs",          "FilePath": "log\_{date}.log"        },        "LogLevel": {          "RoundTheCode.HostedServices.Web.Controllers": "Information",          "RoundTheCode.HostedServices.Web.HostedServices": "Information"        }      }    },    ...  } |

**Instance of Logger Provider and Configuration**

The last thing we need to do is to create a singleton instance of our logger provider. We also need to set the options from our configuration.

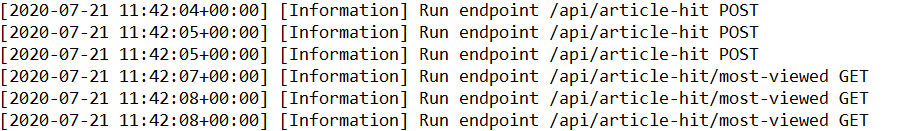
The first thing to do is to set up a method extension that allows us to do this.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | // RoundTheCodeFileLoggerExtensions.cs  public static class RoundTheCodeFileLoggerExtensions  {      public static ILoggingBuilder AddRoundTheCodeFileLogger(this ILoggingBuilder builder, Action<RoundTheCodeFileLoggerOptions> configure)      {          builder.Services.AddSingleton<ILoggerProvider, RoundTheCodeFileLoggerProvider>();          builder.Services.Configure(configure);          return builder;      }  } |

Then, we need to open up our Program class and look inside the CreateHostBuilder method. The Host class has a ConfigureLogging method. Inside that method, we call our AddRoundTheCodeFileLogger method extension, specifying the location of where our options are. In this instance, our options are in Logging, RoundTheCodeFile, Options in appsettings.json.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | // Program.cs  public class Program  {      ...        public static IHostBuilder CreateHostBuilder(string[] args) =>          Host.CreateDefaultBuilder(args)              .ConfigureWebHostDefaults(webBuilder =>              {                  webBuilder.UseStartup<Startup>();              })              .ConfigureLogging((hostBuilderContext, logging) =>              {                  logging.AddRoundTheCodeFileLogger(options =>                  {                      hostBuilderContext.Configuration.GetSection("Logging").GetSection("RoundTheCodeFile").GetSection("Options").Bind(options);                  });              });  } |

And that's it. We have set up logging to be outputted to text files.



**Background Tasks**

Logging is important in all areas of your application, but it's even more important in background tasks. That's because it's very hard to keep tracks on a task that is hidden. It might have stopped working, or it might be producing an error that is unknown to you.

Watch back our live stream where we demonstrate how easy it is to add logging onto a background task and how you can write these logs to text files.

You can also [**download the source code**](https://www.roundthecode.com/dotnet-samples/create-your-own-file-logger) if you wish to try it out for yourselves.