**How To Start Logging With .NET**

Logging is an important part of every application life cycle. Having a good logging system becomes a key feature that helps developers, sysadmins, and support teams to understand and solve appearing problems.

Every log message has an associated log level. The log level helps you understand the severity and urgency of the message. Usually, each log level has an assigned integer representing the severity of the message.

Though .NET provides an extendable logging system, third-party logging libraries can significantly simplify the logging process and usage of advanced logging practices.

In this tutorial, we are going to use a Console Application in C# for our examples, and cover 4 methods of logging in .NET:

* Logging with built-in tools
* Logging with Serilog
* Logging with NLog
* Logging with log4net

**Prerequisites**

You will need:

* Windows 10 installed.
* Visual Studio installed.

If you don't have a console application project ready, you can start by going through the following setup. Otherwise you can skip and go directly to the logging options.

**[The fastest log  
search on the planet](https://betterstack.com/logs?utm_content=upsell&utm_medium=guides&utm_source=community&utm_term=how-to-start-logging-with-net)**

[Better Stack lets you see inside any stack, debug any issue, and resolve any incident.](https://betterstack.com/logs?utm_content=upsell&utm_medium=guides&utm_source=community&utm_term=how-to-start-logging-with-net)

[Explore more](https://betterstack.com/logs?utm_content=upsell&utm_medium=guides&utm_source=community&utm_term=how-to-start-logging-with-net)

**How to create a project**

To get started, you need to create a new project. You can do it in several ways in Visual Studio. The first one is to select **Create a new project** in Visual Studio start window.

The second one, If the Visual Studio IDE is already open, you can follow the path on the top menu bar **File > New > Project**.

The last way we are going to show you is to use the shortcut: Ctrl + Shift + N.

**How to install dependencies**

Before starting to work on the application, you need to install some dependency packages. Visual Studio provides multiple ways to use the NuGet Package Manager. In the tutorial we will use the Package Manager Console.

So, the first thing you have to do is to open the Package Manager Console. You can do it using **Tools > NuGet Package Manager > Package Manager Console**. Alternatively, you can sequentially  press ALT, V, E, and O.

In the console that opens, run the Install-Package command with the package name as an argument, like in the example below.

Install-Package PackageName

**Option 1 — Logging With Built-in Tools**

.NET provides a built-in logging API with great functionality. However, this API also requires using logging providers, such as Console, Debug, EventLog, and others.

**Step 1 — Installing Dependencies In The Package Manager Console**

In the Package Manager Console, run the following commands one by one.

Install-Package Microsoft.Extensions.Logging

Install-Package Microsoft.Extensions.Logging.Console

Install-Package Microsoft.Extensions.Logging.Debug

The Microsoft.Extensions.Logging package includes built-in logging API. The Microsoft.Extensions.Logging.Console and Microsoft.Extensions.Logging.Debug include the logging providers.

**Step 2 — Creating A Logger Factory**

Preparatory to creating a logger, we have to create a logging factory. In the logging factory, you can specify logging targets, the minimum level to log, and other configuration options.

For the application, we're going to use the default .NET log levels system. The system is represented by an enum in the Microsoft.Extensions.Logging namespace and consists of the 7 levels:

* **Critical** — used for reporting about errors that are forcing shutdown of the application.
* **Error** — used for logging serious problems occurring during execution of the program.
* **Warning**  — used for reporting non-critical unusual behaviour.
* **Info** — used for informative messages highlighting the progress of the application for sysadmins and end users.
* **Debug** — used for debugging messages with extended information about application processing.
* **Trace** — used for tracing the code.
* **None** — not used for writing log messages. Specifies that a logging category should not write any messages.

You can find additional information about it [on the Microsoft pages](https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.logging.loglevel?view=dotnet-plat-ext-5.0).

Let's create a logger factory in the Program.cs file.

Program.cs

using System;

using Microsoft.Extensions.Logging;

public class Program

{

static void Main(string[] args)

{

// create a logger factory

var loggerFactory = LoggerFactory.Create(

builder => builder

// add console as logging target

.AddConsole()

// add debug output as logging target

.AddDebug()

// set minimum level to log

.SetMinimumLevel(LogLevel.Debug)

);

}

}

In the code snippet above, we have specified 2 logging targets for the factory: the console and the debug output. Also, we have set the minimum log level.

**Step 3 — Creating A Logger**

At the moment you are ready to create a logger using the factory. The logger will be an instance of the Microsoft.Extensions.Logging.ILogger.

Program.cs

using System;

using Microsoft.Extensions.Logging;

public class Program

{

static void Main(string[] args)

{

// create a logger factory

var loggerFactory = LoggerFactory.Create(

builder => builder

// add console as logging target

.AddConsole()

// add debug output as logging target

.AddDebug()

// set minimum level to log

.SetMinimumLevel(LogLevel.Debug)

);

// create a logger

var logger = loggerFactory.CreateLogger<Program>();

}

}

In the loggerFactory.CreateLogger<Program>() method call, we have specified the class, where the logger will be used — Program. The full class name will be displayed in your logs to help you understand the messages.

**Step 4 — Logging**

To demonstrate how the logger works, we will log 6 messages. According to the logger factory configuration, the minimum log level is Debug, so the trace logs must be omitted.

The following code should be written in the Program.cs file:

Program.cs

using System;

using Microsoft.Extensions.Logging;

public class Program

{

static void Main(string[] args)

{

// create a logger factory

var loggerFactory = LoggerFactory.Create(

builder => builder

// add console as logging target

.AddConsole()

// add debug output as logging target

.AddDebug()

// set minimum level to log

.SetMinimumLevel(LogLevel.Debug)

);

// create a logger

var logger = loggerFactory.CreateLogger<Program>();

// logging

logger.LogTrace("Trace message");

logger.LogDebug("Debug message");

logger.LogInformation("Info message");

logger.LogWarning("Warning message");

logger.LogError("Error message");

logger.LogCritical("Critical message");

}

}

Now, let's build and run the program. You can simply do this by pressing CTRL + F5.

After the execution, your console's output should look like:

Output

dbug: Program[0]

Debug message

info: Program[0]

Info message

warn: Program[0]

Warning message

fail: Program[0]

Error message

crit: Program[0]

Critical message

Also, if you open the debug output using **Debug > Windows > Output**, your output should contain the following messages:

Output

Program: Debug: Debug message

Program: Information: Info message

Program: Warning: Warning message

Program: Error: Error message

Program: Critical: Critical message

**Option 2 — Logging With Serilog**

Serilog is an easy-to-set-up logging library for .NET with a clear API. It is useful in the simplest small applications as well as in large and complex ones. Due to its rich configuration abilities, you can use it in all your projects.

**Step 1 — Installing Dependencies**

Before starting work on the application, you need to install the dependency packages. Visual Studio provides multiple ways to use the NuGet Package Manager. In the tutorial we will use the Package Manager Console.

So, the first thing you have to do is to open the Package Manager Console. You can do it using **Tools > NuGet Package Manager > Package Manager Console**. Alternatively, you can sequentially  press ALT, V, E, and O.

In the Package Manager Console, run the following commands one by one.

Install-Package Serilog

Install-Package Serilog.Sinks.Console

Install-Package Serilog.Sinks.Debug

The Serilog package contains the types. While the Serilog.Sinks namespace includes logging providers, such as Console, Debug, File, HTTP, and others. We've decided to install Console and Debug sinks.

The list of available sinks with additional information about each of them is located [on the Serilog GitHub](https://github.com/serilog/serilog/wiki/Provided-Sinks).

**Step 2 — Creating A Logger**

Now, you are ready to create a logger with basic configuration.

For the application, we're going to use the Serilog's levels system. The consists of the 6 levels:

* **Fatal** — used for reporting about errors that are forcing shutdown of the application.
* **Error** — used for logging serious problems occurred during execution of the program.
* **Warning**  — used for reporting non-critical unusual behaviour.
* **Information** — used for informative messages highlighting the progress of the application for sysadmins and end users.
* **Debug** — used for debugging messages with extended information about application processing.
* **Verbose** — the noisiest level, used for tracing the code.

The default Level, if no minimum level is specified, is the information level.

The following code should be written in the Program.cs file:

Program.cs

using System;

using Serilog;

using System;

using Serilog;

class Program

{

static void Main(string[] args)

{

// create a logger

using var logger = new LoggerConfiguration()

// add console as logging target

.WriteTo.Console()

// add debug output as logging target

.WriteTo.Debug()

// set minimum level to log

.MinimumLevel.Debug()

.CreateLogger();

}

}

**Step 3 — Logging**

To demonstrate how the logger works, we will log 6 messages. According to the logger configuration, the minimum log level is debug, so the verbose logs must be omitted.

The following code should be written in the Program.cs file:

Program.cs

using System;

using Serilog;

class Program

{

static void Main(string[] args)

{

// create a logger

using var logger = new LoggerConfiguration()

// add console as logging target

.WriteTo.Console()

// add debug output as logging target

.WriteTo.Debug()

// set minimum level to log

.MinimumLevel.Debug()

.CreateLogger();

// logging

logger.Verbose("Verbose message");

logger.Debug("Debug message");

logger.Information("Info message");

logger.Warning("Warning message");

logger.Error("Error message");

logger.Fatal("Fatal message");

}

}

Now, let's build and run the program. You can simply do this by pressing CTRL + F5.

After the execution, your console's output should look like:

Output

[06:23:18 DBG] Debug message

[06:23:18 \*\*INF\*\*] Info message

[06:23:18 WRN] Warning message

[06:23:18 ERR] Error message

[06:23:18 FTL] Fatal message

Also, if you open the debug output using **Debug > Windows > Output**, your output should contain the same messages.

**Option 3 — NLog**

NLog is a logging framework for .NET. It has rich log routing and management capabilities and helps you a lot with producing and managing logs. NLog supports structured logs, multiple logging targets, and everything a modern logging framework should support.

**Step 1 — Installing Dependencies**

In the Package Manager Console, run the following commands one by one.

Install-Package NLog.Extensions.Logging

Install-Package Microsoft.Extensions.DependencyInjection

The NLog.Extensions.Logging package is the root package of NLog.

The Microsoft.Extensions.DependencyInjection is the default implementation of dependency injection in .NET. You can read more about it [in the documentation](https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.dependencyinjection?view=dotnet-plat-ext-5.0).

**Step 2 — Creating A Config**

NLog provides 2 methods of configuration: using a configuration file and programmatic configuration. In the following example, we're going to use the programmatic method.

Advanced information about the NLog configuration can be found [in the documentation](https://github.com/NLog/NLog/wiki/Tutorial#configure-nlog-targets-for-output).

Let's create our config. For the application, we're going to use the NLog's default log levels system. The consists of the 6 levels:

* **Fatal** — used for reporting about errors that are forcing shutdown of the application.
* **Error** — used for logging serious problems occurred during execution of the program.
* **Warning**  — used for reporting non-critical unusual behavior.
* **Information** — used for informative messages highlighting the progress of the application for sysadmins and end users.
* **Debug** — used for debugging messages with extended information about application processing.
* **Trace** — the noisiest level, used for tracing the code.

The following code should be written in the Program.cs file:

Program.cs

using System;

using NLog;

class Program

{

static void Main(string[] args)

{

// create a configuration instance

var config = new NLog.Config.LoggingConfiguration();

// create a console logging target

var logConsole = new NLog.Targets.ConsoleTarget();

// create a debug output logging target

var logDebug = new NLog.Targets.OutputDebugStringTarget();

// send logs with levels from Info to Fatal to the console

config.AddRule(NLog.LogLevel.Info, NLog.LogLevel.Fatal, logConsole);

// send logs with levels from Debug to Fatal to the console

config.AddRule(NLog.LogLevel.Debug, NLog.LogLevel.Fatal, logDebug);

// apply the configuration

NLog.LogManager.Configuration = config;

}

}

In the code snippet above, we have specified 2 logging targets for the factory: the console and the debug output. Also, we have set the minimum log level for each of them.

**Step 3 — Creating A Logger**

The logger creating in NLog is pretty straightforward. You can get the current class logger in one line.

The following code should be written in the Program.cs file:

Program.cs

using System;

using NLog;

class Program

{

static void Main(string[] args)

{

// create a configuration instance

var config = new NLog.Config.LoggingConfiguration();

// create a console logging target

var logConsole = new NLog.Targets.ConsoleTarget();

// create a debug output logging target

var logDebug = new NLog.Targets.OutputDebugStringTarget();

// send logs with levels from Info to Fatal to the console

config.AddRule(NLog.LogLevel.Info, NLog.LogLevel.Fatal, logConsole);

// send logs with levels from Debug to Fatal to the console

config.AddRule(NLog.LogLevel.Debug, NLog.LogLevel.Fatal, logDebug);

// apply the configuration

NLog.LogManager.Configuration = config;

// create a logger

var logger = LogManager.GetCurrentClassLogger();

// logging

logger.Trace("Trace message");

logger.Debug("Debug message");

logger.Info("Info message");

logger.Warn("Warning message");

logger.Error("Error message");

logger.Fatal("Fatal message");

}

}

**Step 4 — Logging**

To demonstrate how the logger works, we will log 6 messages. According to the logger configuration, Debug and Trace logs must be omitted in the console, while in the debug output only trace messages must be omitted.

The following code should be written in the Program.cs file:

Program.cs

using System;

using NLog;

class Program

{

static void Main(string[] args)

{

// create a configuration instance

var config = new NLog.Config.LoggingConfiguration();

// create a console logging target

var logConsole = new NLog.Targets.ConsoleTarget();

// create a debug output logging target

var logDebug = new NLog.Targets.OutputDebugStringTarget();

// send logs with levels from Info to Fatal to the console

config.AddRule(NLog.LogLevel.Info, NLog.LogLevel.Fatal, logConsole);

// send logs with levels from Debug to Fatal to the console

config.AddRule(NLog.LogLevel.Debug, NLog.LogLevel.Fatal, logDebug);

// apply the configuration

NLog.LogManager.Configuration = config;

// create a logger

var logger = LogManager.GetCurrentClassLogger();

// logging

logger.Trace("Trace message");

logger.Debug("Debug message");

logger.Info("Info message");

logger.Warn("Warning message");

logger.Error("Error message");

logger.Fatal("Fatal message");

}

}

Now, let's build and run the program. You can simply do this by pressing CTRL + F5.

After the execution, your console's output should look like:

Output

2021-04-19 07:11:25.9385|INFO|Program|Info message

2021-04-19 07:11:25.9769|WARN|Program|Warning message

2021-04-19 07:11:25.9769|ERROR|Program|Error message

2021-04-19 07:11:25.9769|FATAL|Program|Fatal message

Also, if you will open the debug output using **Debug > Windows > Output**, your output should contain the same messages and an extra one — the debug message.

Output

2021-04-19 07:11:25.9187|DEBUG|Program|Debug message

2021-04-19 07:11:25.9385|INFO|Program|Info message

2021-04-19 07:11:25.9769|WARN|Program|Warning message

2021-04-19 07:11:25.9769|ERROR|Program|Error message

2021-04-19 07:11:25.9769|FATAL|Program|Fatal message

**Option 4 — Logging With Log4net**

The log4net is a logging framework for .NET based on Apache log4j. It supports multiple logging targets, structured output, and logging hierarchy. Also, the log4net has [great documentation](https://logging.apache.org/log4net/release/features.html), a lot of related materials, and a big developer community.

**Step 1 — Installing Dependencies**

In the Package Manager Console, run the following commands one by one.

Install-Package log4net

**Step 2 — Creating A Config**

The log4net package provides 2 methods of configuration: using a configuration file and programmatic configuration. In the following example, we're going to use the programmatic method.

Advanced information about the log4net configuration can be found [in the documentation](https://logging.apache.org/log4net/release/manual/configuration.html).

Let's create our config. For the application, we're going to use the log4net's log levels system. The consists of the 5 levels:

* **Fatal** — used for reporting about errors that are forcing shutdown of the application.
* **Error** — used for logging serious problems occurred during execution of the program.
* **Warn**  — used for reporting non-critical unusual behaviour.
* **Info** — used for informative messages highlighting the progress of the application for sysadmins and end users.
* **Debug** — used for debugging messages with extended information about application processing.

The following code should be written in the Program.cs file:

Program.cs

using System;

using log4net;

using log4net.Config;

using log4net.Appender;

using log4net.Repository.Hierarchy;

class Program

{

static void Main(string[] args)

{

// create a hierarchy for configuration

var hierarchy = (Hierarchy)LogManager.GetRepository();

// create console appender

var consoleAppender = new ConsoleAppender();

// add appender

hierarchy.Root.AddAppender(consoleAppender);

// apply the configuration

BasicConfigurator.Configure(hierarchy);

}

}

**Step 3 — Creating A Logger**

The logger creating in log4net is pretty straightforward. You can get the logger in one line.

The following code should be written in the Program.cs file:

Program.cs

using System;

using log4net;

using log4net.Config;

using log4net.Appender;

using log4net.Repository.Hierarchy;

class Program

{

static void Main(string[] args)

{

// create a hierarchy for configuration

var hierarchy = (Hierarchy)LogManager.GetRepository();

// create console appender

var consoleAppender = new ConsoleAppender();

// add appender

hierarchy.Root.AddAppender(consoleAppender);

// apply the configuration

BasicConfigurator.Configure(hierarchy);

// create a logger instance

var logger = LogManager.GetLogger(typeof(Program));

}

}

**Step 4 — Logging**

To demonstrate how the logger works, we will log 5 messages. According to the logger configuration, all log messages must be displayed in the console.

The following code should be written in the Program.cs file:

Program.cs

using System;

using log4net;

using log4net.Config;

using log4net.Appender;

using log4net.Repository.Hierarchy;

class Program

{

static void Main(string[] args)

{

// create a hierarchy for configuration

var hierarchy = (Hierarchy)LogManager.GetRepository();

// create console appender

var consoleAppender = new ConsoleAppender();

// add appender

hierarchy.Root.AddAppender(consoleAppender);

// apply the configuration

BasicConfigurator.Configure(hierarchy);

// create a logger instance

var logger = LogManager.GetLogger(typeof(Program));

// logging

logger.Debug("Debug message");

logger.Info("Info message");

logger.Warn("Warning message");

logger.Error("Error message");

logger.Fatal("Fatal message");

}

}

Now, let's build and run the program. You can simply do this by pressing CTRL + F5.

After the execution, your console's output should look like:

Output

753 [1] DEBUG Program (null) - Debug message

1151 [1] INFO Program (null) - Info message

1152 [1] WARN Program (null) - Warning message

1153 [1] ERROR Program (null) - Error message

1153 [1] FATAL Program (null) - Fatal message

**Conclusion**

Proper logging can greatly assist in the support and development of your application. This may seem like a daunting task, but .NET has good built-in tools and many fast and configurable logging libraries.

Now developing and maintaining your .NET applications will be much easier! For further resources check [.NET application monitoring tools](https://betterstack.com/community/comparisons/dotnet-application-monitoring-tools/)