log4net Tutorial

Tim Corey, 3 Sep 2015

Learn how to use log4net without fear. Stop copying config files and figure out how to make log4net do exactly what you want.

Download source code - 218 KB

Introduction

One of the greatest logging tools out there for .NET is log4net. This software is the gold standard for how logging should be done. It is simple, powerful, and extensible. The best part is that it is part of the FOSS community. What could be better? The one thing I see that is, in my opinion, a bit lacking is a straight-forward tutorial on how to use log4net. The documentation covers, in depth, how to use the software, but it is a bit obscure. Basically, if you already know what log4net can do and you just want to know the syntax, the documentation is for you. The tutorials out there usually cover one piece or one type of system. I am hoping to at least add to the tutorials that are out there, and maybe I might just provide you with a complete tutorial including questions to some of the problems I encountered. The below examples and information are based upon the documentation provided by the log4net group.

Video Tutorial

As a supplement to this article, I have created a video tutorial on YouTube that will go over the material in this article. While this article is complete in itself and it has been updated as the versions of log4net have changed, I have found that some people need to not only read about something but also see it (I am one of those). As a result, the video Application Logging in C#: The log4net tutorial was created. I hope you enjoy it, as well as the rest of this article.

The Basics

There are a three parts to log4net. There is the configuration, the setup, and the call. The configuration is typically done in the *app.config* or *web.config* file. We will go over this in depth below. If you desire more flexibility through the use of a separate configuration file, see the section titled "Getting Away from app.config". Either way you choose to store the configuration information, the code setup is basically a couple of lines of housekeeping that need to be called in order to set up and instantiate a connection to the logger. Finally, the simplest part is the call itself. This, if you do it right, is very simple to do and the easiest to understand.

Logging Levels

There are seven logging levels, five of which can be called in your code. They are as follows (with the highest being at the top of the list):

1. OFF - nothing gets logged (cannot be called)
2. FATAL
3. ERROR

4. WARN
5. INFO
6. DEBUG
7. ALL - everything gets logged (cannot be called)

These levels will be used multiple times, both in your code as well as in the config file. There are no set rules on what these levels represent (except the first and last).

The Configuration

The standard way to set up a log4net logger is to utilize either the *app.config* file in a desktop application or the *web.config* file in a web application. There are a few pieces of information that need to be placed in the *config* file in order to make it work properly with log4net. These sections will tell log4net how to configure itself. The settings can be changed without re-compiling the application, which is the whole point of a *config* file.

Root

You need to have one root section to house your top-level logger references. These are the loggers that inherit information from your base logger (root). The only other thing that the root section houses is the minimum level to log. Since everything inherits from the root, no appenders will log information below that specified here. This is an easy way to quickly control the logging level in your application. Here is an example with a default level of INFO (which means DEBUG messages will be ignored) and a reference to two appenders that should be enabled under root:

```
<root>
  <level value="INFO"/>
  <appender-ref ref="FileAppender"/>
  <appender-ref ref="ConsoleAppender" />
</root>
```

Additional Loggers

Sometimes you will want to know more about a particular part of your application. log4net anticipated this by allowing you to specify additional logger references beyond just the root logger. For example, here is an additional logger that I have placed in our config file to log to the console messages that occur inside the `OtherClass` class object:

```
<logger name="Log4NetTest.OtherClass">
  <level value="DEBUG"/>
  <appender-ref ref="ConsoleAppender"/>
</logger>
```

Note that the logger name is the full name of the class including the namespace. If you wanted to monitor an entire namespace, it would be as simple as listing just the namespace you wanted to monitor. I would recommend against trying to re-use appenders in multiple loggers. It can be done, but you can get some unpredictable results.

ConfigSections

In a config file where there will (potentially) be more information stored beyond just the log4net configuration information, you will need to specify a section to identify where the log4net configuration is housed. Here is a sample section that specifies that the configuration information will be stored under the XML tag "`log4net`":

```
<configSections>
  <section name="log4net"

    type="log4net.Config.Log4NetConfigurationSectionHandler, log4net"/>
</configSections>
```

Appender (General)

An appender is the name for what logs the information. It specifies where the information will be logged, how it will be logged, and under what circumstances the information will be logged. While each appender has different parameters based upon where the data will be going, there are some common elements. The first is the name and type of the appender. Each appender must be named (anything you want) and have a type assigned to it (specific to the type of appender desired). Here is an example of an appender entry:

```
<appender name="ConsoleAppender" type="log4net.Appender.ConsoleAppender">
```

Layout

Inside of each appender must be a layout section. This may be a bit different depending on the type of appender being used, but the basics are the same. You need a type that specifies how the data will be written. There are multiple options, but the one that I suggest you use is the pattern layout type. This will allow you to specify how you want your data written to the data repository. If you specify the pattern layout type, you will need a sub-tag that specifies a conversion pattern. This is the pattern by which your data should be written to the data repository. I will give a more detailed description of your options for the conversion patterns, but for now, here is an example of the layout tag with the pattern layout specified:

```
<layout type="log4net.Layout.PatternLayout">
      <conversionPattern value="%date [%thread] %-5level %logger [%ndc]
    - %message%newline"/>
    </layout>
```

Conversion Patterns

As I mentioned above, the conversion pattern entry is used for the pattern layout to tell the appender how to store the information. There are many different keywords that can be used in these patterns, as well as string literals. Here I will specify what I think are the most useful and important ones. The full list can be found in the log4net documentation.

- `%date` - Outputs the date using the local time zone information. This date can be formatted using the curly braces and a layout pattern such as %date{MMMM dd, yyyy HH:mm:ss, fff} to output the value of "January 01, 2011 14:15:43, 767". However, it is suggested that you use one of the log4net date formatters (ABSOLUTE, DATE, or ISO8601) since they offer better performance.
- `%utcdate` - This is the same as the `%date` modifier, but it outputs in universal time. The modifiers for date/time all work the same way.
- `%exception` - If an exception is passed in, it will be entered and a new line will be placed after the exception. If no exception is passed in, this entry will be ignored and no new line will be put in. This is usually placed at the end of a log entry, and usually a new line is placed before the exception as well.
- `%level` - This is the level you specified for the event (DEBUG, INFO, WARN, etc.).
- `%message` - This is the message you passed into the log event.
- `%newline` - This is a new line entry. Based upon the platform you are using the application on, this will be translated into the appropriate new line character(s). This is the preferred method to enter a new line and it has no performance problems compared to the platform-specific operators.
- `%timestamp` - This is the number of milliseconds since the start of the application.
- `%thread` - This will give you the name of the thread that the entry was made on (or the number if the thread is not named).

Beyond these are a few more that can be very useful but should be used with caution. They have negative performance implications and should be used with caution. The list includes:

- `%identity` - This is the user name of the current user using the `Principal.Identity.Name` method.
- `%location` - Especially useful if you are running in Debug mode, this tells you where the log method was called (line number, method, etc.). However, the amount of information will decrease as you operate in Release mode depending on what the system can access from the compiled code.
- `%line` - This is the line number of the code entry (see the note above on the location issues).
- `%method` - This is the method that calls the log entry (see the note above on the location issues).
- `%username` - This outputs the value of the `WindowsIdentity` property.

You may notice that some config files have letters instead of names. These have been depreciated in favor of whole word entries like I have specified above. Also, while I won't cover it in depth here, note that each of these entries can be formatted to fit a certain width. Spaces can be added (to either side) and values can be truncated in order to fit inside of fixed-width columns. The basic syntax is to place a numeric value or values between the % sign and the name. Here are the modifiers:

- `X` - Specifies the minimum number of characters. Anything that has fewer characters will have spaces placed on the left of the value to equal 20 characters including the message. For example, `%10message` will give you "`hi`".
- `-X` - Same as above, only the spaces will be placed on the right. For example, `%-10message` will give you "`hi `".
- `.X` - Specifies the maximum number of characters. The important thing to note is that this will truncate the beginning of the string, not the end. For example, `%.10message` will give me "`rror entry`" if the string passed in was "`Error entry`".

You can put all of this together with something like this: "`%10.20message`", which would specify that if the message isn't ten characters long, put spaces on the left to fill it out to ten characters, but if the message is more than 20 characters long, cut off the beginning to make it only 20 characters.

Filters

Filters are another big part of any appender. With a filter, you can specify which level(s) to log and you can even look for keywords in the message. Filters can be mixed and matched, but you need to be careful when doing so. When a message fits inside the criteria for a filter, it is logged and the processing of the filter is finished. This is the biggest gotcha of a filter. Therefore, ordering of the filters becomes very important if you are doing a complex filter.

StringMatchFilter

The string match filter looks to find a specific string inside of the information being logged. You can have multiple string match filters specified. They work like `OR` statements in a query. The filter will look for the first string, then the second, etc., until a match is found. However, the important thing to note here is that not finding a match to a specified string does not exclude an entry (since it may proceed to the next string match filter). This means, however, that you may encounter a time where there are no matches found. In that case, the default action is to log the entry. So, at the end of a string match filter set, it is necessary to include a deny all filter (see below) to deny the entry from being logged if a match has not been made. Here is an example of how to filter for entries that have *test* in their message:

```
<filter type="log4net.Filter.StringMatchFilter">
  <stringToMatch value="test" />
</filter>
```

LevelRangeFilter

A level range filter tells the system to only log entries that are inside of the range specified. This range is inclusive, so in the below example, events with a level of INFO, WARN, ERROR, or FATAL will be logged, but DEBUG events will be ignored. You do not need the deny all filter after this entry since the deny is implied.

```
<filter type="log4net.Filter.LevelRangeFilter">
  <levelMin value="INFO" />
  <levelMax value="FATAL" />
</filter>
```

LevelMatchFilter

The level match filter works like the level range filter, only it specifies one and only one level to capture. However, it does not have the deny built into it so you will need to specify the deny all filter after listing this filter.

```
<filter type="log4net.Filter.LevelMatchFilter">
  <levelToMatch value="ERROR"/>
</filter>
```

DenyAllFilter

Here is the entry that, if forgotten, will probably ensure that your appender does not work as intended. The only purpose of this entry is to specify that no log entry should be made. If this were the only filter entry, then nothing would be logged. However, its true purpose is to specify that nothing more should be logged (remember, anything that has already been matched has been logged).

```
<filter type="log4net.Filter.DenyAllFilter" />
```

Appenders

Each type of appender has its own set of syntax based upon where the data is going. The most unusual ones are the ones that log to databases. I will list a few of the ones that I think are most common. However, given the above information, you should be able to use the examples given online without any problems. The log4net site has some great examples of the different appenders. As I have said before, I used the log4net documentation extensively and this area was no exception. I usually copy their example and then modify it for my own purposes.

Console Appender

I use this appender for testing usually, but it can be useful in production as well. It writes to the output window, or the command window if you are using a console application. This particular filter outputs a value like "2010-12-26 15:41:03,581 [10] WARN Log4NetTest.frmMain - This is a WARN test." It will include a new line at the end.

```
<appender name="ConsoleAppender" type="log4net.Appender.ConsoleAppender">
  <layout type="log4net.Layout.PatternLayout">
    <conversionPattern value="%date{ABSOLUTE}
    [%thread] %level %logger - %message%newline"/>
  </layout>
  <filter type="log4net.Filter.StringMatchFilter">
    <stringToMatch value="test" />
  </filter>
  <filter type="log4net.Filter.DenyAllFilter" />
</appender>
```

File Appender

This appender will write to a text file. The big differences to note here are that we have to specify the name of the text file (in this case, it is a file named *mylogfile.txt* that will be stored in the same location as the executable), we have specified that we should append to the file (instead of overwriting it), and we have specified that the `FileAppender` should use the Minimal Lock which will make the file usable by multiple appenders.

```
<appender name="FileAppender" type="log4net.Appender.FileAppender">
  <file value="mylogfile.txt" />
  <appendToFile value="true" />
  <lockingModel type="log4net.Appender.FileAppender+MinimalLock" />
  <layout type="log4net.Layout.PatternLayout">
    <conversionPattern value="%date [%thread] %level %logger - %message%newline" />
  </layout>
  <filter type="log4net.Filter.LevelRangeFilter">
    <levelMin value="INFO" />
    <levelMax value="FATAL" />
  </filter>
</appender>
```

Rolling File Appender

This is an appender that should be used in place of the file appender whenever possible. The purpose of the rolling file appender is to perform the same functions as the file appender but with the additional option to only store a certain amount of data before starting a new log file. This way, you won't need to worry about the logs on a system filling up over time. Even a small application could overwhelm a file system given enough time writing to a text file if the rolling option were not used. In this example, I am logging in a similar fashion to the file appender above, but I am specifying that the log file should be capped at 10MB and that I should keep up to 5 archive files before I start deleting them (oldest gets deleted first). The archives will be named with the same name as the file, only with a dot and the number after it (example: *mylogfile.txt.2* would be the second log file archive). The `staticLogFileName` entry ensures that the current log file will always be named what I specified in the file tag (in my case, *mylogfile.txt*).

```
<appender name="RollingFileAppender" type="log4net.Appender.RollingFileAppender">
  <file value="mylogfile.txt" />
  <appendToFile value="true" />
  <rollingStyle value="Size" />
  <maxSizeRollBackups value="5" />
  <maximumFileSize value="10MB" />
```

```
  <staticLogFileName value="true" />
  <layout type="log4net.Layout.PatternLayout">
    <conversionPattern value="%date [%thread] %level %logger - %message%newline" />
  </layout>
</appender>
```

ADO.NET Appender

Here is the tricky one. This specific example writes to SQL, but you can write to just about any database you want using this pattern. Note that the `connectionType` is basically a connection string, so modifying it is simple. The `commandText` specified is a simple query. You can modify it to any type of `INSERT` query that you want (or Stored Procedure). Notice that each parameter is specified below and mapped to a log4net variable. The size can be specified to limit the information placed into the parameter. This appender is a direct copy from the log4net example. I take no credit for it. I simply use it as an example of what can be done.

Quick note: *If you find that your ADO.NET appender is not working, check the* `bufferSize` *value. This value contains the number of log statements that log4net will cache before writing them all to SQL. The example on the log4net website has a* `bufferSize` *of 100, which means you will probably freak out in testing when nothing is working. Change the* `bufferSize` *value to 1 to make the logger write every statement when it comes in.*

For this example and more, go to the following URL: http://logging.apache.org/log4net/release/config-examples.html.

```
<appender name="AdoNetAppender" type="log4net.Appender.AdoNetAppender">
  <bufferSize value="100" />
  <connectionType value="System.Data.SqlClient.SqlConnection,
   System.Data, Version=1.0.3300.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" />
  <connectionString value="data source=[database server];
    initial catalog=[database name];integrated security=false;
    persist security info=True;User ID=[user];Password=[password]" />
  <commandText value="INSERT INTO Log ([Date],[Thread],[Level],[Logger],
    [Message],[Exception]) VALUES (@log_date, @thread, @log_level,
    @logger, @message, @exception)" />
  <parameter>
    <parameterName value="@log_date" />
    <dbType value="DateTime" />
    <layout type="log4net.Layout.RawTimeStampLayout" />
  </parameter>
  <parameter>
    <parameterName value="@thread" />
    <dbType value="String" />
    <size value="255" />
    <layout type="log4net.Layout.PatternLayout">
      <conversionPattern value="%thread" />
    </layout>
  </parameter>
  <parameter>
    <parameterName value="@log_level" />
    <dbType value="String" />
    <size value="50" />
    <layout type="log4net.Layout.PatternLayout">
      <conversionPattern value="%level" />
    </layout>
  </parameter>
  <parameter>
    <parameterName value="@logger" />
    <dbType value="String" />
    <size value="255" />
    <layout type="log4net.Layout.PatternLayout">
      <conversionPattern value="%logger" />
    </layout>
  </parameter>
  <parameter>
    <parameterName value="@message" />
    <dbType value="String" />
    <size value="4000" />
    <layout type="log4net.Layout.PatternLayout">
      <conversionPattern value="%message" />
    </layout>
  </parameter>
  <parameter>
    <parameterName value="@exception" />
```

```
    <dbType value="String" />
    <size value="2000" />
    <layout type="log4net.Layout.ExceptionLayout" />
  </parameter>
</appender>
```

The Code

Once you have a reference to the log4net DLL in your application, there are three lines of code that you need to know about. The first is a one-time entry that needs to be placed outside of your class. I usually put it right below my `using` statements in the *Program.cs* file. You can copy and paste this code since it will probably never need to change (unless you do something unusual with your *config* file). Here is the code:

```
[assembly: log4net.Config.XmlConfigurator(Watch = true)]
```

The next entry is done once per class. It creates a variable (in this case called "`log`") that will be used to call the log4net methods. This code is also code that you can copy and paste (unless you are using the Compact Framework). It does a `System.Reflection` call to get the current class information. This is useful because it allows us to use this code all over but have the specific information passed into it in each class. Here is the code:

```
private static readonly log4net.ILog log = log4net.LogManager.GetLogger
    (System.Reflection.MethodBase.GetCurrentMethod().DeclaringType);
```

The final code piece is the actual call to log some piece of information. This can be done using the following code:

```
log.Info("Info logging");
```

Notice that you can add an optional parameter at the end to include the exception that should be logged. Include the entire exception object if you want to use this option. The call is very similar, and it looks like this:

```
log.Error("This is my error", ex);
```

`ex` is the exception object. Remember that you need to use the *%exception* pattern variable in your appender to actually capture this exception information.

Logging Extra Data

Using the basic configuration in log4net usually includes enough information for a typical application. However, sometimes you want to record more information in a standard way. For example, if you use the ADO.NET appender, you may want to add a field for application user name instead of just including it in the message field. There isn't a conversion pattern that matches up with the application user name. However, you can use the Context properties to specify custom properties that can be accessed in the appenders. Here is an example of how to set it up in code:

```
log4net.GlobalContext.Properties["testProperty"] = "This is my test property information";
```

There are a couple of things to notice. First, I named the property "`testProperty`". I could have named it anything. However, be careful because if you use a name that is already in use, you may overwrite it. This leads into the second thing to note. I referenced the `GlobalContext`, but there are four contexts that can be utilized. They are based upon the threading. Global is available anywhere in the application where Thread, Logical Thread, and Event restrict the scope further and further. You can use this to store different information based upon the context of where the logger was called. However, if you have two properties with the same name, the one that is in the narrower scope will win. Looking at our first point again, we can see the issue that this might cause. If we declare a `GlobalContext` property that has the same property name as an existing `ThreadContext`, we may not see the property value we expect because of the existing value. For this reason, I would suggest developing your own naming scheme that will not conflict with anyone else's names.

Here is an example of how to capture this property in our appender:

```
<layout type="log4net.Layout.PatternLayout">
  <conversionPattern value="%date{ABSOLUTE} [%thread] %level
                            %logger - %message%newlineExtra Info: %property{
```

```
                        testProperty}%newline%exception"/>
</layout>
```

For more information on the different Contexts, reference the log4net documentation on the topic here: http://logging.apache.org/log4net/release/manual/contexts.html.

Getting Away from app.config/web.config

You may come across a time when you want to use a separate file to store the log4net configuration information. In fact, you might find this to be the optimal way to store the configuration information, since you could keep copies of your different standard configurations on hand to drop into your projects. This could cut down on development time and allow you to standardize your logging information. To set this up, you need to change only two parts of your app. The first thing you need to do is save the configuration in a different file. The format will be the same, as will how it is laid out. The only thing that will really change in the layout is that it isn't in the middle of your *app.config* or *web.config* file. The second change you need to make is in that one setup call in your application. You need to add information on where the file is, like so:

```
[assembly: log4net.Config.XmlConfigurator(ConfigFile =
                "MyStandardLog4Net.config", Watch = true)]
```

There is also the possibility of simply choosing a different extension for this file by using "`ConfigFileExtension`" instead of "`ConfigFile`" in the line above. If you do that, you need to name your config file to be your assembly name (including extension), and it needs to have the extension you specify. Here is an example with a more visual explanation:

```
[assembly: log4net.Config.XmlConfigurator(ConfigFileExtension = "mylogger", Watch = true)]
```

In the above example, if our application was *test.exe*, then the configuration file for log4net should be named *text.exe.mylogger*.

VB.NET

For those of you who would like to use log4net in a VB.NET application, there are a few differences that need to be noted. The *config* file will stay the same, the reference DLL is the same, and the logging calls are exactly the same (just drop off the semicolon at the end), but the setup calls are different. The first setup command is to reference the assembly. This only needs to be placed once somewhere globally in the application (outside of a class). Here is the command:

```
'Standard Configuration
<Assembly: log4net.Config.XmlConfigurator(Watch:=True)>

'Using a config file besides app.config/web.config
<Assembly: log4net.Config.XmlConfigurator(
      ConfigFile:="MyStandardLog4Net.config", Watch:=True)>
```

The next change is in the reference variable that we set up once per class. This change, like the previous one, is just a direct conversion from C# to VB.NET syntax:

```
Private Shared ReadOnly log As log4net.ILog = log4net.LogManager.GetLogger(_
        System.Reflection.MethodBase.GetCurrentMethod().DeclaringType)
```

That little bit of code conversion is all it takes to make this a VB.NET project. I've updated the download to have both the C# project and the VB.NET project in it so that you can try either. I did come across two little issues that might throw you for a loop when you create your own project in VB.NET. The first issue I found was that they hid the framework selection deeper in the properties menu. Remember that we need to change the target framework from ".NET Framework 4 Client Profile" to ".NET Framework 4" in order for log4net to work properly (**Note**: this is no longer the case with the latest version of log4net). In order to find this, open up the project properties page. Under the Compile tab, there is a button at the bottom named "Advanced Compile Options..." Click this and you will be given the option to change your target framework. The other issue I found was that I couldn't add an *app.config* file because it said I already had one. I had to click the "Show all files" button to see my (existing) *app.config* file. Since you need to modify this specific file, make sure you find the right one. Don't forget that you can copy and paste your *config* file from a C# project without issues. *Config* files for log4net are language independent.

Config File Template

While you can look at the example code I have posted to see a config file in action, based upon some of the difficulties people were experiencing, I decided to post a config file template to help readers visualize where each of the config file pieces will go. I have given you a blank template below. I have also labeled each section with which level it is in so that, in case the formatting doesn't make it obvious, you know how each item relates to all the others up and down the tree.

```
<!--This is the root of your config file-->
<configuration> <!-- Level 0 -->
  <!--This specifies what the section name is-->
  <configSections> <!-- Level 1 -->
    <section name="log4net"

      type="log4net.Config.Log4NetConfigurationSectionHandler,
            log4net"/> <!-- Level 2 -->
  </configSections>
  <log4net> <!-- Level 1 -->
    <appender>  <!-- Level 2 -->
      <layout>  <!-- Level 3 -->
        <conversionPattern />  <!-- Level 4 -->
      </layout>
      <filter>  <!-- Level 3 -->
      </filter>
    </appender>
    <root> <!-- Level 2 -->
      <level /> <!-- Level 3 -->
      <appender-ref /> <!-- Level 3 -->
    </root>
    <logger> <!-- Level 2 -->
      <level /> <!-- Level 3 -->
      <appender-ref /> <!-- Level 3 -->
    </logger>
  </log4net>
</configuration>
```

FAQs

1. **Why can't I write to my text file?** log4net runs under the privileges of the active user. Make sure that the active user has rights to create/modify/delete the specified text file.
2. **Why aren't certain events being logged?** Most likely, this is because you have a filter in place. Remember that the root entry can have an overall minimum log filter that specifies no event gets logged below this level. The actual appender can also have a level filter, or you could have the deny all in place too early.
3. **Why am I logging events that I don't want?** Most likely you have set a filter improperly. Either you forgot the deny all filter, or you included a level range filter first.
4. **Why am I getting a compile error?** If you get an error similar to "The referenced assembly "log4net" could not be resolved because it has a dependency on "System.Web, Version=4.0.0.0 ...", then you haven't changed your target framework to ".NET Framework 4". Change that and this error, along with others that cascade from it, will go away. ***Note**: this has been fixed in the new version of log4net. You should no longer need to do this.*
5. **How do I use this in ASP.NET?** Using this information in ASP.NET is the same as using it in a desktop application. The major difference is that the *config* file is the *web.config* file instead of the *app.config* file.
6. **Why can't I get my ADO.NET appender to log anything?** If you look over all of the settings and they look right, chances are that you are experiencing the pain of the `bufferSize` configuration. Change the `bufferSize` to 1 and it will attempt to log every message you send right away. If this still does not work, the issue is your configuration.

Conclusion

I hope you have found this tutorial to be useful. I believe that I have covered all of the information you need to get started using log4net without fear. Let me know if you would like me to expound on any area listed above or if you have an issue using log4net. For a complete example, look at the source code that I have provided. It is a working example of how to use log4net. You can use this as a testing platform for your *config* files. Use it to make sure you are logging entries the way you expect, before copying the *config* file information over to your production application.

History

- 26th December, 2010: Initial version.
- 31st December, 2010: Added VB.NET section, various grammatical changes, and minor clarifications.
- 1st February, 2011: Added advanced information on loggers, properties. and alternate log files, clarified complete config file layout, and added information on ADO.NET gotchas.
- 19th June, 2012: Minor updates to reflect the new version of log4net.
- 4th September, 2015: Added the video tutorial section

License

About the Author

Tim Corey

Software Developer (Senior) DeGarmo
United States

I am currently a Senior Software Developer at a company in Illinois called DeGarmo. My primary skills are in .NET, SQL, JavaScript, and other web technologies although I have worked with PowerShell, C, and Java as well.

In my previous positions, I have worked as a lead developer, professor and IT Director. As such, I have been able to develop software on a number of different types of systems and I have learned how to correctly oversee the overall direction of technology for an organization. I've developed applications for everything from machine automation to complete ERP systems.

I enjoy taking hard subjects and making them easy to understand for people unfamiliar with the topic.

Comments and Discussions

270 messages have been posted for this article Visit **https://www.codeproject.com/Articles/140911/log-net-Tutorial** to post and view comments on this article, or click **here** to get a print view with messages.