Published on **ONDotNet.com** (http://www.ondotnet.com/)
http://www.ondotnet.com/pub/a/dotnet/2003/06/16/log4net.html
See this if you're having trouble printing code examples

Using log4net

by Nauman Leghari
06/16/2003

Introduction

Logging is an essential tool in every developer's arsenal. It helps the developer to identify problems faster by showing the state of an application at any given point. It is important after deployment, when all that the poor system admins have are the logs that are generated by your application. So it is absolutely necessary to be equipped with a logging framework which is easy to set up, easy to use, and extensible. With this in mind, we will be discussing `log4net`, an open source logging and tracing framework. The only prerequisite for this article is to know how to program in .NET using C#, although the concepts are applicable to programmers using VB.NET or any other .NET language.

About `log4net`

`log4net`, as I said earlier, is an open source project and is the port of the famous `log4j` project for Java. It is an excellent piece of work, started by a team at www.neoworks.com, but it would not have been possible without the contributions made by the community. `log4net` provides many advantages over other logging systems, which makes it a perfect choice for use in any type of application, from a simple single-user application to a complex multiple-threaded distributed application using remoting. The complete features list can be viewed here. It can be downloaded from the web site under the Apache license. The latest version at this writing is 1.2.0 beta 7, upon which this article is based. The changes in this release are listed here.

You can see from the feature document that this framework is released for four different platforms. There are separate builds for Microsoft .NET Framework, Microsoft .NET Compact Framework, Mono 0.23, and SSCLI 1.0. There are different levels of support provided with each framework, the details of which are documented here. This version of `log4net` is provided with NAnt build scripts. To compile the framework, you can execute the *build.cmd* file from the root directory where you extracted the zipped file. The *log4net.sln* file in the *<log4net-folder>\src* directory is the solution file for `log4net` source, whereas the examples are provided in a separate solution file in *<log4net-folder>\examples\net\1.0*. The samples are provided in C#, VB.NET, VC++.NET, and even in JScript.NET. Some of the samples have their configuration files in the project's root folder, so in order to run those samples you need to manually move them with project's executable file. The API documentation is provided in the *<log4net-folder>\doc\sdk\net* directory.

The Structure of `log4net`

`log4net` is built using the layered approach, with four main components inside of the framework. These are Logger, Repository, Appender, and Layout.

Logger

The *Logger* is the main component with which your application interacts. It is also the component that generates the log messages.

Generating a log message is different than actually showing the final output. The output is showed by the Layout component, as we will see later.

The logger provides you with different methods to log any message. You can create multiple loggers inside of your application. Each logger that you instantiate in your class is maintained as a "named entity" inside of the `log4net` framework. That means that you don't need to pass around the Logger instance between different classes or objects to reuse it. Instead, you can call it with the name anywhere in the application. The loggers maintained inside of the framework follow a certain organization. Currently, the `log4net` framework uses the hierarchical organization. This hierarchy is similar to the way we define namespaces in .NET. For example, say there are two loggers, defined as `a.b.c` and `a.b`. In this case, the logger `a.b` is said to be the ancestor of the logger `a.b.c`. Each logger inherits properties from its parent logger. At the top of the hierarchy is the default logger, which is also called the root logger, from which all loggers are inherited. Although this namespace-naming scheme is preferred in most scenarios, you are allowed to name your logger as you would like.

The `log4net` framework defines an interface, `ILog`, which is necessary for all loggers to implement. If you want to implement a custom logger, this is the first thing that you should do. There are a few examples in the */extension* directory to get you started.

The skeleton of the `ILog` interface is shown below:

```
public interface ILog
{
  void Debug(object message);
  void Info(object message);
  void Warn(object message);
  void Error(object message);
  void Fatal(object message);

  // There are overloads for all of the above methods which
  // supports exceptions. Each overload in that case takes an
  // addition parameter of type Exception like the one below.
  void Debug(object message, Exception ex);

  // ...
  // ...
  // ...

  // The Boolean properties are used to check the Logger's
  // level (as we'll see Logging Levels in the next section)
  bool isDebugEnabled;
  bool isInfoEnabled;

  // other boolean properties for each method
}
```

From this layer, the framework exposes a class called `LogManager`, which manages all loggers. It has a `GetLogger()` method that retrieves the logger for us against the name we provided as a parameter. It will also create the logger for us if it is not already present inside of the framework.

```
log4net.ILog log = log4net.LogManager.GetLogger("logger-name");
```

Most often, we define the class type as the parameter to track the name of the class in which we are

logging. The name that is passed is prefixed with all of the log messages generated with that logger. The type of class can be passed in by name using the `typeof(Classname)` method, or it can be retrieved through reflection by the following statement:

```
System.Reflection.MethodBase.GetCurrentMethod().DeclaringType
```

Despite the long syntax, the latter is used in the samples for its portability, as you can copy the same statement anywhere to get the class in which it is used.

Logging Levels

As you can see in the `ILog` interface, there are five different methods for tracing an application. Why do we need all of these different methods? Actually, these five methods operate on different levels of priorities set for the logger. These different levels are defined as constants in the `log4net.spi.Level` class.

You can use any of the methods in your application, as appropriate. But after using all of those logging statements, you don't want to have all of that code waste CPU cycles in the final version that is deployed. Therefore, the framework provides seven levels and their respective Boolean properties to save a lot of CPU cycles. The value of `Level` can be one of the following:

Table 1. Different `Level`s of a Logger

Level	Allow Method	Boolean Property	**Value**
`OFF`			Highest
`FATAL`	`void Fatal(...);`	`bool IsFatalEnabled;`	
`ERROR`	`void Error(...);`	`bool IsErrorEnabled;`	
`WARN`	`void Warn(...);`	`bool IsWarnEnabled;`	
`INFO`	`void Info(...);`	`bool IsInfoEnabled;`	
`DEBUG`	`void Debug(...);`	`bool IsDebugEnabled;`	
`ALL`			Lowest

In the `log4net` framework, each logger is assigned a priority level (which is one of the values from the table above) through the configuration settings. If a logger is not assigned a `Level`, then it will try to inherit the `Level` value from its ancestor, according the hierarchy.

Also, each method in the `ILog` interface has a predefined value of its level. As you can see in Table 1, the `Info()` method of the `ILog` interface has the `INFO` level. Similarly, the `Error()` method has the `ERROR` level, and so on. When we use any of these methods, the `log4net` framework checks the method level against the level of the logger. The logging request is said to be enabled if the logger's level is greater than or equal to the level of the logging method.

For example, let's say you create a logger object and set it to the level of `INFO`. The framework then sets the individual Boolean properties for that logger. The level checking is performed when you call any of the logging methods.

```
Logger.Info("message");
Logger.Debug("message");
Logger.Warn("message");
```

For the first method, the level of method `Info()` is equal to the level set on the logger (`INFO`), so the request passes through and we get the output, "message."

For the second method, the level of the method `Debug()` is less than that of the logger (see Table 1). There, the request is disabled or refused and you get no output.

Similarly, you can easily conclude what would have happened in the third line.

There are two special `Level`s defined in Table 1. One is `ALL`, which enables all requests, and the other is `OFF`, which disables all requests.

You can also explicitly check the level of the logger object through the Boolean properties.

```
if (logger.IsDebugEnabled)
{
  Logger.Debug("message");
}
```

Repository

The second layer is responsible for maintaining the organization of loggers. By organization, I am talking about the logical structure of the loggers inside of the framework. Before the current version of `log4net`, the framework only supported the hierarchical organization. As we discussed earlier, this hierarchical nature is an implementation of the repository and is defined in the `log4net.Repository.Hierarchy` namespace. To implement a `Repository`, it is necessary to implement the `log4net.Repository.ILoggerRepository` interface. But instead of directly implementing this interface, another class, `log4net.Repository.LoggerRepositorySkeleton`, is provided to work as the base class; e.g., the hierarchical repository is implemented by the `log4net.Repository.Hierarchy.Hierarchy` class.

If you are a normal developer only using the `log4net` framework instead of extending it, then you would probably not use any of these `Repository` classes in your code. Instead, you would use the `LogManager` class, as described earlier, to automatically manage the repositories and the loggers.

Appender

Any good logging framework should be able to generate output for multiple destinations, such as outputting the trace statements to the console or serializing it into a log file. `log4net` is a perfect match for this requirement. It uses a component called *Appender* to define this output medium. As the name suggests, these components append themselves to the Logger component and relay the output to an output stream. You can append multiple appenders to a single logger. There are several appenders provided by the `log4net` framework; the complete list of appenders provided by the `log4net` framework can be found here.

With all of these appenders provided, there is not much need for writing your own, but if you wish to, you can start by inheriting the `log4net.Appender.AppenderSkeleton` class, which works as an adapter between your class and the `IAppender` interface.

Appender Filters

An Appender defaults to pass all logging events to the Layout. *Appender Filters* can be used to select events by different criteria. There are several filters defined under the `log4net.Filter` namespace.

By using a filter, you can either filter a range of level values, or filter out any log message with a particular string. We'll see filters in action later in our example. More information about filters is provided in the API documentation.

Layout

The *Layout* component is used to display the final formatted output to the user. The output can be shown in multiple formats, depending upon the layout we are using. It can be linear or an XML file. The layout component works with an appender. There is a list of different layouts in the API documentation. You cannot use multiple layouts with an appender. To create your own layout, you need to inherit the `log4net.Layout.LayoutSkeleton` class, which implements the `ILayout` interface.

Using `log4net` in Your Application

Before you start logging your application, you need to heat up the `log4net` engine. Technically, this means that you need to configure the three components that we discussed earlier. There are two different methods by which you can specify the configuration: you can either define them in a separate configuration file, or you can place them inside of your code, configuring it programmatically.

The first method is always recommended, for the following reasons.

- You can change the settings without recompiling the source files.
- You can change the settings even when your application is running. This is very important in web application and remote application scenarios.

Considering the importance of the first method, we'll see it first.

Using a Configuration File

The configuration settings required are put into either of the following files:

1. In the application config file (*AssemblyName.config* or *web.config*).
2. Into your own file. The filename could be anything you like, or it could be name of the assembly with a different extension concatenated onto it (such as *AppName.exe.xyz*).

The `log4net` framework looks for the configuration file in the file path relative to the application's base directory defined by the `AppDomain.CurrentDomain.BaseDirectory` property. The only thing that the `log4net` framework searches for inside of the configuration file is the `<log4net>` tag. A complete sample configuration file is shown below:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="log4net"
      type="log4net.Config.Log4NetConfigurationSectionHandler,
            log4net-net-1.0"
    />
  </configSections>

  <log4net>

    <logger name="testApp.Logging">
      <level value="DEBUG"/>
```

```
    </logger>

    <root>
      <level value="WARN" />
      <appender-ref ref="LogFileAppender" />
      <appender-ref ref="ConsoleAppender" />
    </root>

    <appender name="LogFileAppender"
             type="log4net.Appender.FileAppender" >
      <param name="File" value="log-file.txt" />
      <param name="AppendToFile" value="true" />
      <layout type="log4net.Layout.PatternLayout">
        <param name="Header" value="[Header]\r\n"/>
        <param name="Footer" value="[Footer]\r\n"/>
        <param name="ConversionPattern"
           value="%d [%t] %-5p %c [%x] <%X{auth}> - %m%n"
         />
      </layout>
      <filter type="log4net.Filter.LevelRangeFilter">
        <param name="LevelMin" value="DEBUG" />
        <param name="LevelMax" value="WARN" />
      </filter>
    </appender>

    <appender name="ConsoleAppender"
              type="log4net.Appender.ConsoleAppender" >
      <layout type="log4net.Layout.PatternLayout">
        <param name="ConversionPattern"
           value="%d [%t] %-5p %c [%x] <%X{auth}> - %m%n"
        />
      </layout>
    </appender>

  </log4net>
</configuration>
```

You can copy the above file to use in any application, but it is always better to know what constitutes the configuration file. The `<section>` entry inside of the `<configSection>` tag is only necessary if you are using the application's configuration file. Otherwise, only the text inside of the `<log4net>` tag is required. It is not a requirement to maintain the sequence of individual tags; I am only putting it this way to maintain the flow of describing things. Taking each tag individually, we start with the `<logger>` element.

<Logger>

```
<logger name="testApp.Logging">
  <level value="DEBUG"/>
  <appender-ref ref="LogFileAppender" />
  <appender-ref ref="ConsoleAppender" />
</logger>
```

The `<logger>` element defines the settings for an individual logger. Then by calling `LogManager.GetLogger(...)`, you can retrieve the same logger by the name. You can also define the appenders to use with that logger through the `<appender-ref>` tag. `<appender-ref>` defines a reference to an appender which is actually defined anywhere else.

<root>

```
<root>
  <level value="WARN" />
  <appender-ref ref="LogFileAppender" />
  <appender-ref ref="ConsoleAppender" />
</root>
```

The `<root>` tag is next to the logger tag. All loggers in the hierarchy are children of the root logger; therefore the framework uses the properties defined here if there are no loggers explicitly defined in the configuration file. After knowing this, we can also tell that the `<logger>` tag that we see above is not necessary. Inside of the `<root>` tag, the default values are defined. Both the level value and the appender list can be put in here. The default value of `LEVEL`, if not defined anywhere else, is set to `DEBUG`. Obviously, the individual setting for a logger in the `<logger>` tag overrides the settings in the root tag for that particular logger. In the case of an appender, the `<logger>` tag will inherit all of the appenders defined by its ancestor. This default behavior can be changed by explicitly setting the *additivity* attribute for the `<logger>` tag to `false`.

```
<logger name="testApp.Logging" additivity="false">
</logger>
```

This attribute is set to true by default. The `<root>` tag is not necessary, but recommended.

`<appender>`

```
<appender name="LogFileAppender"
          type="log4net.Appender.FileAppender" >
  <param name="File" value="log-file.txt" />
  <param name="AppendToFile" value="true" />
  <layout type="log4net.Layout.PatternLayout">
    <param name="Header" value="[Header]\r\n" />
    <param name="Footer" value="[Footer]\r\n"/>
    <param name="ConversionPattern"
      value="%d [%t] %-5p %c [%x] &lt;%X{auth}&gt; - %m%n"
    />
  </layout>
  <filter type="log4net.Filter.LevelRangeFilter">
    <param name="LevelMin" value="DEBUG" />
    <param name="LevelMax" value="WARN" />
  </filter>
</appender>
```

The appenders listed either in the `<root>` tag or in the individual `<logger>` tag are defined individually using the `<appender>` tag. The basic format of the `<appender>` tag is defined above. It uses the appender name and maps it to the class that defines that appender. Other important things to see here are the tags inside of the `<appender>` element. The `<param>` tag varies with different appenders. Here, to use the `FileAppender`, you need a file name that you can define as a parameter. To complete the picture, a `Layout` is defined inside of the `<appender>` tag. The layout is declared in its own `<layout>` tag. The `<layout>` element defines the layout type (`PatternLayout` in the example) and the parameters that are required by that layout (as in the pattern string used by the `PatternLayout` class).

The `Header` and `Footer` tags provide the text to print before and after a logging session. The details of configuring each appender are further described in the documentation here, where you can see individual appender section as examples.

The last thing is the `<filter>` tag in the `Appender` element. It defines the filter to apply to a specific

Appender. In this example, we are applying the LevelRangeFilter, which extracts only those messages that fall between the levels defined between the LevelMin and LevelMax parameters. Similarly, other tags are defined, as appropriate. Multiple filters can be applied to an appender, which then work in a pipeline in the sequence in which they are ordered. Other filters and information on using them can be found in the log4net SDK documents.

These are the necessary elements that we needed to initialize the log4net framework for our application. Now that we have created the configuration file, it's time to link to it from our application.

By default, every standalone executable assembly defines its own configuration settings. The log4net framework uses the log4net.Config.DOMConfiguratorAttribute on the assembly level to set the configuration file. There are three properties for this attribute.

1. ConfigFile: The property is only used if we are defining the <log4net> tag into our own configuration file.
2. ConfigFileExtension: If we are using the application compiled assembly with a different extension, then we need to define the extension here.
3. Watch (Boolean): This is the property by which the log4net system decides whether to watch the file for runtime changes or not. If the value is true, then the FileSystemWatcher class is used to monitor the file for change, rename, and delete notifications.

```
[assembly:log4net.Config.DOMConfigurator(ConfigFile="filename",
  ConfigFileExtension="ext",Watch=true/false)]
```

The log4net framework will consider the application's configuration file if you do not define either the ConfigFile or ConfigFileExtension attribute. These attributes are mutually exclusive. We also need to keep it in mind that the DOMConfigurator attribute is necessary and can be defined as follows, with no parameters:

```
[assembly: log4net.Config.DOMConfigurator()]
```

There is another technique that saves you from having to use the attributes. It uses the DOMConfigurator class inside of the code to load the configuration file provided in the parameter. This method takes a FileInfo object instead of a file name. This method has the same effect as loading the file through the attribute, as shown previously.

```
log4net.Config.DOMConfigurator.Configure(
  new FileInfo("TestLogger.Exe.Config"));
```

There is another method, ConfigureAndWatch(..), in the DOMConfigurator class, to configure the framework to watch the file for any changes.

The above step concludes everything related to configuration. Next, the following two steps are required in our code to use the logger.

1. Create a new logger or get the logger you already created. It uses the setting defined in the configuration file. If this particular logger is not defined in the configuration file, then the framework uses the logger's hierarchy to gather different parameters from its ancestors and, lastly, from the root logger.

```
Log4net.ILog log = log4net.LogManager.GetLogger("logger-name");
```

2. Use the log object to call any of the logger methods. You can also check the level of the logger through the `IsXXXEnabled` Boolean variables before calling the methods to boost performance.

```
if (log.IsDebugEnabled) log.Debug("message");
if (log.IsInfoEnabled) log.Info("message);
//….
```

Configuring `log4net` Programmatically

Sometimes we are in the mood to code as quickly as possible without getting into configuration files. Normally, that happens when we are trying to test something. In that case, you have another way to do the configuration. All of the long configuration files that we saw in the previous section can be defined programmatically using a few lines of code. See the following code:

```
// using a FileAppender with a PatternLayout
log4net.Config.BasicConfigurator.Configure(
  new log4net.Appender.FileAppender(
     new log4net.Layout.PatternLayout("%d
       [%t]%-5p %c [%x] &lt;%X{auth}&gt; - %m%n"),"testfile.log"));

// using a FileAppender with an XMLLayout
log4net.Config.BasicConfigurator.Configure(
  new log4net.Appender.FileAppender(
    new log4net.Layout.XMLLayout(),"testfile.xml"));

// using a ConsoleAppender with a PatternLayout
log4net.Config.BasicConfigurator.Configure(
  new log4net.Appender.ConsoleAppender(
    new log4net.Layout.PatternLayout("%d
      [%t] %-5p %c [%x] &lt;%X{abc}&gt; - %m%n")));
/
/ using a ConsoleAppender with a SimpleLayout
log4net.Config.BasicConfigurator.Configure(
  new log4net.Appender.ConsoleAppender(new
    log4net.Layout.SimpleLayout()));
```

You can see that while it is easy to code here, you can't configure settings for individual loggers. All of the settings that are defined here are applied to the root logger.

The `log4net.Config.BasicConfigurator` class uses its static `Configure` method to set an `Appender` object. The `Appender` constructor, in turn, requires the `Layout` object. Other parameters are respective to the type of component you are using.

You can also use `BasicConfigurator.Configure()` without any parameter to show the output using `ConsoleAppender` with a specific `PatternLayout`, as follows:

Code

```
log4net.Config.BasicConfigurator.Configure();
```

Output

```
0 [1688] DEBUG log1 A B C - Test
20 [1688] INFO log1 A B C - Test
```

Now that the application is configured, you can write the logging code, as shown in the previous section.

Logging in a Multithreaded Application

One of the most noticeable features of `log4net` is its support for multithreaded applications. This helps you in scenarios where your application is simultaneously accessed by multiple clients. Therefore, to trace requests from different clients, you need a mechanism to identify different clients in your logging framework. This mechanism is provided in `log4net` through two different methods, Nested Diagnostic Context (NDC) and Mapped Diagnostic Context (MDC).

Nested Diagnostic Context (NDC)

NDC uses a stack per thread to identify different clients. The stack's `Push()` method is used to set any value that identifies the client. It is the developer's responsibility to put in a unique value for each client. To constrain NDC into a certain block of code, the developer can use the `"using"` statement to make his task easier, because it automatically pops the respective value from the stack.

```
using(log4net.NDC.Push("clientid")
{
  log.Info("message"); // output: "clientid - message"
} // here the value is popped from the stack
```

NDC class can also be used without the using block.

```
log4net.NDC.Push("clientid"); // context started here
…. // put client aware log messages here
log4net.NDC.Pop(); // context ended here
```

The framework provides a special conversion term, "`%x`," to display the NDC value on the stack using the `PatternLayout`.

```
<layout type="log4net.Layout.PatternLayout,log4net">
  <param name="ConversionPattern" value="%x" />
</layout>
```

If you push multiple values into the stack, then all of those values are concatenated in the output.

If you are using the `XMLLayout` class as the layout for your appender, then you automatically have the NDC value as a `CDATA` section inside of the `<log4net:NDC>` tag.

```
<log4net:NDC><![CDATA[A B C]]></log4net:NDC>
```

Mapped Diagnostic Context (MDC)

Instead of using a stack, the MDC class uses a map to store individual user information. It could not be used inside of the using block; therefore, we have to use it in a `Get()`/`Set()` combination to manipulate the map values. Values can be deleted from the map using the `Remove()` method. Similar to NDC, MDC also works on a per-thread model, and requires the conversion term inside of the pattern string, if you are using the `PatternLayout`. For MDC, the term is "`%X`" (capital X) with the

key concatenated to the character in curly braces. In the following example, `%X{clientid}` is replaced with the value for the key `clientid`.

```
<layout type="log4net.Layout.PatternLayout">
  <param name="ConversionPattern"
         value="%X{clientid}"
  />
</layout>
```

Logging in ASP.NET

Using `log4net` with ASP.NET is similar to the other applications. Here, you can put the configuration attributes inside of the *global.asax* file, as well as in any of the WebForms files. Putting it in *global.asax* is far easier to remember.

More with `log4net`

There is a lot left to explore in the `log4net` framework. New features are also integrating into the framework quite frequently. Once you get started using the information provided in this article, the next step would be to experiment with the samples provided with the sources. In order to follow the updates on the framework, I'll try to use the forum to post any changes. I would like to acknowledge Nicko Cadell from www.Neoworks.com for reviewing this article and helping me with some technical details.

Resources

- `log4net` – log4net.sourceforge.net
- "How to configure `log4net` with WebServices"
- Complete Features List: log4net.sourceforge.net/release/1.2.0.30507/doc/features.html
- Release Notes: log4net.sourceforge.net/release/1.2.0.30507/releasenotes.html
- Appenders List and Example Configurations: log4net.sourceforge.net/release/1.2.0.30507/doc/manual/example-config-appender.html
- Framework Support Document: log4net.sourceforge.net/release/1.2.0.30507/doc/manual/framework-support.html
- FAQ: log4net.sourceforge.net/release/latest/doc/manual/faq.html
- User List @ sourceforge: log4net-users@lists.sourceforge.net
- `log4net` manual: log4net.sourceforge.net/release/latest/doc/manual/introduction.html

Nauman Leghari

Return to ONDotnet.com