Common Infrastructure Libraries for .NET

Reference Documentation

Version 2.0

Last Updated: April 26, 2009

1. Common Logging 1
1.1. Introduction 1
1.2. Upgrading from previous versions 1
1.2.1. Upgrading to 2.0 1
1.3. Using Common.Logging API 2
1.4. Configuring Logging 4
1.4.1. Declarative Configuration 4
1.4.2. Configuring Logging in your code 4
1.5. Logging Adapters 5
1.5.1. NoOpLoggerFactoryAdapter 5
1.5.2. ConsoleOutLoggerFactoryAdapter 5
1.5.3. TraceLoggerFactoryAdapter 5
1.5.4. Log4NetLoggerFactoryAdapter 6
1.5.5. NLogLoggerFactoryAdapter 7
1.5.6. Enterprise Library 3.1 Logging Adapter 8
1.5.7. Enterprise Library 4.1 Logging Adapter 9
1.6. Advanced Logging Tasks 9
1.6.1. Integrating with System.Diagnostics.Trace 9
1.6.2. Implementing a custom FactoryAdapter 10
1.6.3. Bridging logging systems 10

Chapter 1. Common Logging

1.1. Introduction

There are a variety of logging implementations for .NET currently in use, log4net, Enterprise Library Logging, NLog, to name the most popular. The downside of having differerent implementation is that they do not share a common interface and therefore impose a particular logging implementation on the users of your library. To solve this dependency problem the Common.Logging library introduces a simple abstraction to allow you to select a specific logging implementation at runtime.

The library is based on work done by the developers of IBatis.NET and it's usage is inspired by log4net. Many thanks to the developers of those projects! The library is available for .NET 1.0, 1.1, and 2.0 with both debug and strongly signed release assemblies.

The base logging library, Common.Logging, provides the base logging interfaces that appear in your code and also include simple console and trace based logger implementations. The libraries are located under bin/net/<framework-version>/debug or release.

The following logging systems are supported out of the box:

- System.Console
- System.Diagnostics.Trace
- Log4Net 1.2.9
- Log4Net 1.2.10 (higher version by assembly version redirect)
- NLog
- Enterprise Library 3.1 Logging
- Enterprise Library 4.1 Logging

There are two enterprise log4net implementations, one for log4net 1.2.9 and another for log4net 1.2.10. The need for two log4net versions is due to the fact that each is signed with a different strong key making assembly redirection impossible.

Note that it is not the intention of this library to be a replacement for the many fine logging libraries that are out there. The API is incredibly minimal and will very likely stay that way. Only use this library if you truly need to support multiple logging APIs.

1.2. Upgrading from previous versions

1.2.1. Upgrading to 2.0

The new version of Common.Logging assembly is 100% binary backwards compatible to previous versions. Either you rebuild your project against the new version or simply specifiy an assembly version redirect:

```
<configuration>
  <runtime>
     <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
        <dependentAssembly>
           <assemblyIdentity name="Common.Logging" publicKeyToken="af08829b84f0328e" culture=""/>
           <bindingRedirect oldVersion="0.0.0.0-65535.65535.65535.65535" newVersion="2.0.0.0"/>
           <!-- reference specific file: -->
```

```
          <!-- codeBase version="2.0.0.0" href="../../../build/net/2.0/release/Common.Logging.dll"/ -->
       </dependentAssembly>
     </assemblyBinding>
  </runtime>
</configuration>
```

Those who implemented their own Common.Logging.ILoggerFactoryAdapter and Common.Logging.ILog interfaces, need to update their code to the extended interfaces coming with version 2.0. For convenience Common.Logging comes with a couple of support classes, making this task as easy as possible as described in Implementing a custom FactoryAdapter

1.3. Using Common.Logging API

Usage of the Logging API is fairly simple. First you need to obtain a logger from the LogManager and call the appropriate logging method:

```
using Common.Logging;
...
ILog log = LogManager.GetCurrentClassLogger();
log.Debug("hello world");
```

It is also possible to obtain a logger by name:

```
ILog log = LogManager.GetLogger("mylogger");
```

When working on NET 3.0 or higher, a logger offers a convenient way to log information while paying as little performance penalty as possible. Sometimes evaluating log messsage arguments migth be costly. These costs even hit you, when the logging is turned off, just because the log call is in your code. In this case one usually writes:

```
if (log.IsDebugEnabled)
{
  log.Debug( "my expensive to calculate argument is: {0}", CalculateMessageInfo() );
}
```

Since Common.Logging 2.0 there is a shortcut notation available that allows you to write:

```
log.Debug( m=>m("my expensive to calculate argument is: {0}", CalculateMessageInfo()) );
```

This form is equivalent to the example above and guarantees, that `CalculateMessageInfo()` is only called, when the message really gets logged!

Finally here is the complete interface offered by a logger instance:

```
public interface ILog
{
    // Methods
    void Trace(object message);
    void Trace(object message, Exception exception);
    void Trace(FormatMessageCallback formatMessageCallback);
    void Trace(FormatMessageCallback formatMessageCallback, Exception exception);
    void Trace(IFormatProvider formatProvider, FormatMessageCallback formatMessageCallback);
    void Trace(IFormatProvider formatProvider, FormatMessageCallback formatMessageCallback
                , Exception exception);
    void TraceFormat(string format, params object[] args);
    void TraceFormat(string format, Exception exception, params object[] args);
    void TraceFormat(IFormatProvider formatProvider, string format, params object[] args);
    void TraceFormat(IFormatProvider formatProvider, string format, Exception exception
                    , params object[] args);

    void Debug(object message);
    void Debug(object message, Exception exception);
    void Debug(FormatMessageCallback formatMessageCallback);
    void Debug(FormatMessageCallback formatMessageCallback, Exception exception);
    void Debug(IFormatProvider formatProvider, FormatMessageCallback formatMessageCallback);
    void Debug(IFormatProvider formatProvider, FormatMessageCallback formatMessageCallback
```

```
                , Exception exception);
    void DebugFormat(string format, params object[] args);
    void DebugFormat(string format, Exception exception, params object[] args);
    void DebugFormat(IFormatProvider formatProvider, string format, params object[] args);
    void DebugFormat(IFormatProvider formatProvider, string format, Exception exception
                    , params object[] args);

    void Info(object message);
    void Info(object message, Exception exception);
    void Info(FormatMessageCallback formatMessageCallback);
    void Info(FormatMessageCallback formatMessageCallback, Exception exception);
    void Info(IFormatProvider formatProvider, FormatMessageCallback formatMessageCallback);
    void Info(IFormatProvider formatProvider, FormatMessageCallback formatMessageCallback
              , Exception exception);
    void InfoFormat(string format, params object[] args);
    void InfoFormat(string format, Exception exception, params object[] args);
    void InfoFormat(IFormatProvider formatProvider, string format, params object[] args);
    void InfoFormat(IFormatProvider formatProvider, string format, Exception exception
                    , params object[] args);

    void Warn(object message);
    void Warn(object message, Exception exception);
    void Warn(FormatMessageCallback formatMessageCallback);
    void Warn(FormatMessageCallback formatMessageCallback, Exception exception);
    void Warn(IFormatProvider formatProvider, FormatMessageCallback formatMessageCallback);
    void Warn(IFormatProvider formatProvider, FormatMessageCallback formatMessageCallback
              , Exception exception);
    void WarnFormat(string format, params object[] args);
    void WarnFormat(string format, Exception exception, params object[] args);
    void WarnFormat(IFormatProvider formatProvider, string format, params object[] args);
    void WarnFormat(IFormatProvider formatProvider, string format, Exception exception
                    , params object[] args);

    void Error(object message);
    void Error(object message, Exception exception);
    void Error(FormatMessageCallback formatMessageCallback);
    void Error(FormatMessageCallback formatMessageCallback, Exception exception);
    void Error(IFormatProvider formatProvider, FormatMessageCallback formatMessageCallback);
    void Error(IFormatProvider formatProvider, FormatMessageCallback formatMessageCallback
               , Exception exception);
    void ErrorFormat(string format, params object[] args);
    void ErrorFormat(string format, Exception exception, params object[] args);
    void ErrorFormat(IFormatProvider formatProvider, string format, params object[] args);
    void ErrorFormat(IFormatProvider formatProvider, string format, Exception exception
                    , params object[] args);

    void Fatal(object message);
    void Fatal(object message, Exception exception);
    void Fatal(FormatMessageCallback formatMessageCallback);
    void Fatal(FormatMessageCallback formatMessageCallback, Exception exception);
    void Fatal(IFormatProvider formatProvider, FormatMessageCallback formatMessageCallback);
    void Fatal(IFormatProvider formatProvider, FormatMessageCallback formatMessageCallback
               , Exception exception);
    void FatalFormat(string format, params object[] args);
    void FatalFormat(string format, Exception exception, params object[] args);
    void FatalFormat(IFormatProvider formatProvider, string format, params object[] args);
    void FatalFormat(IFormatProvider formatProvider, string format, Exception exception
                    , params object[] args);

    // Properties
    bool IsDebugEnabled { get; }
    bool IsErrorEnabled { get; }
    bool IsFatalEnabled { get; }
    bool IsInfoEnabled { get; }
    bool IsTraceEnabled { get; }
    bool IsWarnEnabled { get; }
}
```

Since the ILog interface mimics that of the interface used in log4net, migration from log4net is just a matter of changing the 'using' statement.

You can get a reference to an instance of an ILog using the LoggingManager class. Its API is shown below:

```
public sealed class LogManager
{
  public static ILog GetLogger( Type type ) ...
  public static ILog GetLogger( string name ) ...

  public static ILoggerFactoryAdapter Adapter ...
}
```

The Adapter property is used by the framework itself.

1.4. Configuring Logging

There are 2 ways of configuring logging in your application - either declaratively or pro grammatically.

1.4.1. Declarative Configuration

Logging configuration can be done declaratively in your app.config

```
<configuration>
  <configSections>
    <sectionGroup name="common">
      <section name="logging" type="Common.Logging.ConfigurationSectionHandler, Common.Logging" />
    </sectionGroup>
  </configSections>

  <common>
    <logging>
      <factoryAdapter type="Common.Logging.Simple.ConsoleOutLoggerFactoryAdapter, Common.Logging">
        <arg key="level" value="DEBUG" />
        <arg key="showLogName" value="true" />
        <arg key="showDataTime" value="true" />
        <arg key="dateTimeFormat" value="yyyy/MM/dd HH:mm:ss:fff" />
      </factoryAdapter>
    </logging>
  </common>
</configuration>
```

Note

The concrete set of <arg> elements you may specify depends on the FactoryAdapter being used.

Note that if you have installed Common.Logging in the GAC, you will need to specify the fully qualified name of the assembly, i.e. add the Version, Culture, and PublicKeyToken, etc. See the log4net section for an example.

1.4.2. Configuring Logging in your code

You may manually configure logging by setting a LoggerFactoryAdapter in your code.

```
// create properties
NameValueCollection properties = new NameValueCollection();
properties["showDateTime"] = "true";

// set Adapter
Common.Logging.LogManager.Adapter = new Common.Logging.Simple.ConsoleOutLoggerFactoryAdapter(properties);
```

Note

The concrete set of properties you may specify depends on the FactoryAdapter being used.

1.5. Logging Adapters

There are simple out-of-the-box implementations coming with Common.Logging itself. For connecting to log4net, separate adapters do exist.

Note

Be sure to correctly specify the type of the FactoryAdapter in the common logging configuration section and to copy the logging implementation .dlls to your runtime directory. At the moment, if the specified FactoryAdapter type is not found or its dependent libraries, the NoOpLoggerFactoryAdaptor is used by default and you will not see any logging output.

1.5.1. NoOpLoggerFactoryAdapter

This is the default FactoryAdapter if logging is not configured. It simply does nothing.

1.5.2. ConsoleOutLoggerFactoryAdapter

ConsoleOutLoggerFactoryAdapter uses Console.Out for logging output.

Table 1.1. Configuration Properties

Key	Possible Value(s)	Description
level	All Debug Info Warn Error Fatal Off	Defines the global maximum level of logging.
showDateTime	true|false	output timestamp?
showLogName	true|false	output logger name?
dateTimeFormat	any formatstring accepted by DateTime.ToString()	defines the format to be used for output the timestamp. If no format is specified DateTime.ToString() will be used.

1.5.3. TraceLoggerFactoryAdapter

TraceLoggerFactoryAdapter uses System.Diagnostics.Trace for logging output. For viewing it's output you can use any tool that is capable of capturing calls to Win32 `OutputDebugString()` - e.g. the tool "DebugView" from www.sysinternals.com.

Table 1.2. Configuration Properties

Key	Possible Value(s)	Description
level	All Debug Info Warn Error Fatal Off	Defines the global maximum level of logging.
showDateTime	true|false	output timestamp?
showLogName	true|false	output logger name?
dateTimeFormat	any formatstring accepted by DateTime.ToString()	defines the format to be used for output the timestamp. If no format is specified DateTime.ToString() will be used.

1.5.4. Log4NetLoggerFactoryAdapter

There are two implementations, both configured similarly.

- `Common.Logging.Log4Net`

  is linked against log4net 1.2.10.0

- `Common.Logging.Log4Net129`

  is linked against log4net 1.2.9.0

The only difference is in the type specified to the factory adapter. Both Adapters accept the following configuration properties:

Table 1.3. Configuration Properties

Key	Possible Value(s)	Description
configType	FILE FILE-WATCH INLINE EXTERNAL	INLINE will simply call XmlConfigurator.Configure() EXTERNAL expects log4net being configured somewhere else in your code and does nothing. FILE, FILE-WATCH: see property "configFile" below.
configFile	<path to your log4net.config file>	if configType is FILE or FILE-WATCH, the value of "configFile" is passed to XmlConfigurator.Configure (FileInfo) / ConfigureAndWatch(FileInfo) method.

The example below will configure log4net 1.2.10.0 using the file `log4net.config` from your application's root directory by calling `XmlConfigurator.ConfigureAndWatch()`:

```
<configuration>
  <common>
    <logging>
      <factoryAdapter type="Common.Logging.Log4Net.Log4NetLoggerFactoryAdapter, Common.Logging.Log4net">
        <arg key="configType" value="FILE-WATCH" />
        <arg key="configFile" value="~/log4net.config" />
      </factoryAdapter>
    </logging>
  </common>
</configuration>
```

For log4net 1.2.9, change the assembly name `Common.Logging.Log4Net129`.

Another example that shows the log4net configuration 'inline' with the standard application configuration file is shown below.

```
<configuration>
  <configSections>
    <sectionGroup name="common">
      <section name="logging" type="Common.Logging.ConfigurationSectionHandler, Common.Logging" />
    </sectionGroup>

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

```
  <common>
    <logging>
      <factoryAdapter type="Common.Logging.Log4Net.Log4NetLoggerFactoryAdapter, Common.Logging.Log4Net">
        <arg key="configType" value="INLINE" />
      </factoryAdapter>
    </logging>
  </common>

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

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

    <logger name="MyApp.DataAccessLayer">
      <level value="DEBUG" />
    </logger>
  </log4net>
</configuration>
```

Note that if you are using Common.Logging or any of the adapters from the GAC, you will need to specify the full type name, including verison, publickey token etc., as shown below

```
<configSections>
  <sectionGroup name="common">
    <section name="logging" type="Common.Logging.ConfigurationSectionHandler, Common.Logging,
                                  Version=2.0.0.0, Culture=neutral, PublicKeyToken=AF08829B84F0328E"/>
  </sectionGroup>
  <section name="log4net" type="log4net.Config.Log4NetConfigurationSectionHandler, log4net,
                                Version=1.2.10.0, Culture=neutral, PublicKeyToken=1B44E1D426115821"/>
</configSections>
<common>
 <logging>
   <factoryAdapter type="Common.Logging.Log4Net.Log4NetLoggerFactoryAdapter, Common.Logging.Log4Net,
                         Version=2.0.2.2, Culture=neutral, PublicKeyToken=AF08829B84F0328E">
     <arg key="configType" value="FILE-WATCH"/>
     <arg key="configFile" value="~/log4net.config"/>
   </factoryAdapter>
 </logging>
</common>
```

1.5.5. NLogLoggerFactoryAdapter

There is one implementation.

- `Common.Logging.NLog`

  is linked against NLog 1.0.0.505

Table 1.4. Configuration Properties

Key	Possible Value(s)	Description
configType	INLINE FILE	INLINE FILE: see property "configFile" below.
configFile	<path to your NLog.config file>	if configType is FILE, the value of "configFile" is passed to XmlLoggingConfiguration(string) constructor.

The example below will configure NLog using the file `NLog.config` from your application's root directory by calling `XmlLoggingConfiguration(string)`:

```
<configuration>
  <common>
    <logging>
      <factoryAdapter type="Common.Logging.NLog.NLogLoggerFactoryAdapter, Common.Logging.NLog">
        <arg key="configType" value="FILE" />
        <arg key="configFile" value="~/NLog.config" />
      </factoryAdapter>
    </logging>
  </common>
</configuration>
```

Another example that shows the NLog configuration 'inline' with the standard application configuration file is shown below.

```
<configuration>
  <configSections>
    <sectionGroup name="common">
      <section name="logging" type="Common.Logging.ConfigurationSectionHandler, Common.Logging" />
    </sectionGroup>

    <section name="nlog" type="NLog.Config.ConfigSectionHandler, NLog"/>
  </configSections>

  <common>
    <logging>
      <factoryAdapter type="Common.Logging.NLog.NLogLoggerFactoryAdapter, Common.Logging.NLog">
        <arg key="configType" value="INLINE" />
      </factoryAdapter>
    </logging>
  </common>

  <nlog xmlns="http://www.nlog-project.org/schemas/NLog.xsd"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <targets>
      <target name="console" xsi:type="Console" layout="${date:format=HH\:MM\:ss} ${logger} ${message}" />
    </targets>
    <rules>
      <logger name="*" minlevel="Debug" writeTo="console" />
    </rules>
  </nlog>
</configuration>
```

1.5.6. Enterprise Library 3.1 Logging Adapter

There is one implementation located in the assembly Common.Logging.EntLib and is linked against the Microsoft Enterprise Library v 3.1, aka EntLib 3.1. The .dlls for EntLib can not be redistributed so you will need to download EntLib separately.

There are no configuration options for the adapter. Configuration of EntLib logging is done entirely though App.config. The example below shows the basic configuration of the EntLibLoggingAdapter

```
<common>
  <logging>
    <factoryAdapter type="Common.Logging.EntLib.EntLibLoggerFactoryAdapter, Common.Logging.EntLib"/>
  </logging>
</common>
```

Future releases may include configuration of the priority and also the format in which information about an exception is logged. The current priority used is -1, the default.

Note that the following mapping of Common.Logging LogLevel to System.Diagnostics.TraceEventType is used.

Table 1.5. EntLib 3.1 to Common.Logging log level mapping

Common.Logging.LogLevel	System.Diagnostics.TraceEventType
Trace	Verbose
Debug	Verbose
Error	Error
Fatal	Critical
Info	Information
Warn	Warning

1.5.7. Enterprise Library 4.1 Logging Adapter

There is one implementation located in the assembly Common.Logging.EntLib41 and is linked against the Microsoft Enterprise Library v4.1, aka EntLib 4.1. The .dlls for EntLib can not be redistributed so you will need to download EntLib separately.

There are no configuration options for the adapter. Configuration of EntLib logging is done entirely though App.config. The example below shows the basic configuration of the EntLibLoggingAdapter

```
<common>
  <logging>
    <factoryAdapter type="Common.Logging.EntLib.EntLibLoggerFactoryAdapter, Common.Logging.EntLib41"/>
  </logging>
</common>
```

Future releases may include configuration of the priority and also the format in which information about an exception is logged. The current priority used is -1, the default.

Note that the following mapping of Common.Logging LogLevel to System.Diagnostics.TraceEventType is used.

Table 1.6. EntLib 4,1 to Common.Logging log level mapping

Common.Logging.LogLevel	System.Diagnostics.TraceEventType
Trace	Verbose
Debug	Verbose
Error	Error
Fatal	Critical
Info	Information
Warn	Warning

1.6. Advanced Logging Tasks

1.6.1. Integrating with System.Diagnostics.Trace

There are 2 ways of integrating with System.Diagnostics.Trace: Either you redirect all log messages from System.Diagnostics.Trace to Common.Logging or vice versa. Logging from Common.Logging to System.Diagnostics.Trace is done by configuring TraceLoggerFactoryAdapter.

To configure System.Diagnostics.Trace to route its output to the Common.Logging infrastructure, you need to configure a special TraceListener:

```
<configuration>
   <system.diagnostics>
      <trace>
         <listeners>
            <clear />
            <add name="commonLoggingListener"
                 type="Common.Logging.CommonLoggingTraceListener, Common.Logging"
                 initializeData="LogLevel=Trace" />
         </listeners>
      </trace>
   </system.diagnostics>
</configuration>
```

This configuration causes all messages logged via System.Diagnostics.Trace.Write to be logged using a logger instance obtained from LogManager by the "name" attribute and using the LogLevel specified in the "initializeData" attribute.

1.6.2. Implementing a custom FactoryAdapter

f you want to plug in a new, yet unsupported logging library, you need to provide a logger factory adapter that implementes the Common.Logging.ILoggerFactoryAdapter interface. Loggers must implement the Common.Logging.ILog interface.

Important: Any implementation *must* provide a public constructor accepting a NameValueCollection parameter as shown in the example below:

```
public class MyLoggingFactoryAdapter :  ILoggerFactoryAdapter
{
  public MyLoggingFactoryAdapter(NameValueCollection properties)
  {
    // configure according to properties
  }

  public ILog GetLogger(Type type) { ... }

  public ILog GetLogger(string name) { ... }
}
```

For convenience, Common.Logging comes with an abstract base class Common.Logging.Factory.AbstractCachingLoggerFactoryAdapter for easier implemention of factory adapters and Common.Logging.Factory.AbstractLogger for implementing loggers.

1.6.3. Bridging logging systems

In the case your application uses frameworks that are tied to different logging systems, one usually had to find a workaround yourself. Using Common.Logging removes this problem: All integration modules come with plugs to route log messages in 2 directions - either send messages from Common.Logging to the 3rd party logging system, or you can feed messages from that other logging system into Common.Logging.

Let's assume, one of your frameworks uses log4net, another one System.Diagnostics.Trace. You prefer the small but powerful NLog system. First you need to configure log4net to send all log events to Common.Logging:

Example 1.1. Route log4net messages to Common.Logging

```
<log4net>
    <appender name="CommonLoggingAppender"
              type="Common.Logging.Log4Net.CommonLoggingAppender, Common.Logging.Log4Net">
        <layout type="log4net.Layout.PatternLayout, log4net">
            <param name="ConversionPattern" value="%level - %class.%method: %message" />
        </layout>
    </appender>

    <root>
        <level value="ALL" />
        <appender-ref ref="CommonLoggingAppender" />
    </root>
</log4net>
```

To get System.Diagnostics.Trace messages routed to Common.Logging, you need to configure the corresponding CommonLoggingTraceListener:

Example 1.2. Route System.Diagnostics.Trace messages to Common.Logging

```
<system.diagnostics>
  <sharedListeners>
    <add name="Diagnostics"
         type="Common.Logging.Simple.CommonLoggingTraceListener, Common.Logging"
         initializeData="DefaultTraceEventType=Information; LoggerNameFormat={listenerName}.{sourceName}">
      <filter type="System.Diagnostics.EventTypeFilter" initializeData="Information"/>
    </add>
  </sharedListeners>
  <trace>
    <listeners>
      <add name="Diagnostics" />
    </listeners>
  </trace>
</system.diagnostics>
```

Finally you want Common.Logging to output all events to NLog:

Example 1.3. Route Common.Logging messages to NLog

```
<nlog autoReload="true" throwExceptions="true">
 <targets>
  <target name="common.logging"
          type="Common.Logging.NLog.CommonLoggingTarget, Common.Logging.NLog"
          layout="${longdate}|${level:uppercase=true}|${message}"
          filename="C:\temp\${date:format=yyyy-MM-dd}_logA.txt"/>
 </targets>
 <rules>
  <logger name="*" minlevel="Info" writeTo="common.logging" />
 </rules>
</nlog>
```