# .NET Standard 2: Logging, tracing and profiling

## Overview

LogScope is a supplemental library that extends the logging function by implementing logging scope. Current version supports the logging to the system console only but be extensible to support any other logging solutions like [log4net](https://logging.apache.org/log4net/), [NLog](https://nlog-project.org/), etc. To keep things simple, it consists of three components: Provider, Formatter and Manager. Manager creates logging scopes, passing the implementation of the provider ILogProvider and formatter IScopeFormater. Provider implements the actual logging function and formatter oversees formatting message line. The library includes console provider and default formatter.

## Installation

LogScope is available in NuGet package manager: <https://www.nuget.org/packages/DevInstance.LogScope/>

Console:

dotnet add package DevInstance.LogScope

Package manager:

Install-Package DevInstance.LogScope

## Initialization

The first step is using ScopeLogFactory to instantiate a manager. For instance, logging to the console with debug level, manager can be instantiated using:

var manager = ScopeLogFactory.CreateConsoleLogger(LogLevel.DEBUG);

To know more about logging level please see <reference to the documentation>.

Also, you can provide additional parameters the formatter by:

var manager = ScopeLogFactory.CreateConsoleLogger(LogLevel.DEBUG, new DefaultFormaterOptions { ShowTimestamp = true, ShowThreadNumber = true });

If you need to provide custom log provider or formatter:

var manager = ScopeLogFactory.Create(LogLevel.DEBUG, customProvider, customrFormater);

For ASP.NET Core applications or WebAsembly, there are extensions to add manager to Service collection (IServiceCollection):

services.AddConsoleLogging(LogLevel.INFO);

## Simple scenario

The whole idea is around using a “scope”. Scope can be method or a specific part of it. The implementation is based on IDisposable where calling Dispose ends the scope.

To understand how it works, let’s look into the following example. Let’s define a simple class:

class TestClass

{

IScopeLog log;

public TestClass(IScopeManager manager)

{

log = manager.CreateLogger(this);

}

public void MethodFoo()

{

using (var methodScope = log.DebugScope())

{

methodScope.D("Wait for 200 msec");

Thread.Sleep(200);

methodScope.D("Done.");

using (var aScope = methodScope.DebugScope("a-scope"))

{

aScope.D("Inside of a scope");

}

}

}

}

…and instantiate it:

new TestClass(manager).MethodFoo();

Here the result in the log:

-->TestClass:MethodFoo

TestClass:MethodFoo:Wait for 200 msec

TestClass:MethodFoo:Done.

-->TestClass:MethodFoo:a-scope

TestClass:MethodFoo:a-scope:Inside of a scope

<--TestClass:MethodFoo:a-scope, time:0.52 msec

<--TestClass:MethodFoo, time:211.0924 msec

As you can see, there are three scopes in this example: TestClass (created in the constructor), MethodFoo and a local scope “a-scope”. Every logging line contains the “scope path” (e.g “TestClass:MethodFoo:a-scope”) as well as “-->” begin and end “<--” of every scope. Knowing the scope’s being and end can be useful when analyzing asynchronous parts of the code.

## Extensions

One of the goals of this library is to maintain simplicity and keep the code clean. The following snipped “using (var methodScope = log.DebugScope())” would not be a great example but library provides extensions to make logging code lines as short as possible. Please see <reference to extensions>

## Advanced scenario