Unit testing ILogger

By Andrew Hinkle

The C# .NET 5 and greater [ILogger](https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.logging.ilogger?view=dotnet-plat-ext-6.0) interface gives you a nice out of box logging implementation with super useful extension methods, that is until you want to unit test it. The extension methods provide some great helpers while simplifying the formatter dependency, but at the expense of unit testing.

**TLDR**: If an extension method off a public interface needs to create a new instance of any object, then the class of that new instance should be public. I think this should be a standard when creating extension methods. I recommend changing FormattedLogValues from internal to public.

# Goals of this article

1. Review ILogger
2. Review ILogger extensions internal implementation of the formatter
3. Provide my ILogger extension methods
4. Provide my Moq extension methods to verify ILogger methods are called
5. Review a sample unit test
   1. A class logs a request, processes the request, logs the response, returns the response
   2. This logging is important, so we want to ensure that the logging is happening
   3. This unit test verifies the log methods were called as expected
      1. Assume we have integration tests for testing full implementation
6. Review how that unit test would change if the formatter was public

There's a lot here, so let's get started.

[GIT: Penblade/Tips/Tips.ApiMessage.src](https://github.com/penblade/Tips/tree/e562ad1d0c032d34872ff5af14a17e2ea5fb26d0/Tips.ApiMessage/src)

# Reference the current Microsoft source code

When I originally started this research effort, I googled the namespace, and the following link is the first to come up. This is the old source code on GitHub abandoned roughly in 2018 and should not be referenced. It's still the first thing that comes up, so beware, it tripped me up.

<https://github.com/aspnet/Logging/tree/master/src/Microsoft.Extensions.Logging.Abstractions>

This is the current source code on GitHub.

<https://github.com/dotnet/runtime/blob/main/src/libraries/Microsoft.Extensions.Logging.Abstractions>

# Microsoft.Extensions.Logging.ILogger

The important piece here is to note that the formatter is: Func<TState, Exception, string>. TState is a generic object that you pass in, while Exception and string are public. So far everything is accessible.

<https://github.com/dotnet/runtime/blob/main/src/libraries/Microsoft.Extensions.Logging.Abstractions/src/ILogger.cs>

using System;

namespace Microsoft.Extensions.Logging

{

public interface ILogger

{

void Log<TState>(LogLevel logLevel, EventId eventId, TState state, Exception? exception, Func<TState, Exception?, string> formatter);

bool IsEnabled(LogLevel logLevel);

IDisposable BeginScope<TState>(TState state);

}

}

# Microsoft.Extensions.Logging.Abstractions.LoggerExtensions

Let's start with the BeginScope extension method. This is an extension method, so it's static, meaning we can't mock it. That's ok, we get around this by mocking the underlying method call on the interface and verify that it's of the type defined in the extension method when verifying the method was called.

Uh oh, "new FormattedLogValues…", that's an **INTERNAL** class. We can't check the type because it's not accessible. It's an extension method, so we can't inject our own instance. If the class was public, we'd be fine.

<https://github.com/dotnet/runtime/blob/main/src/libraries/Microsoft.Extensions.Logging.Abstractions/src/LoggerExtensions.cs#L410-L421>

public static IDisposable BeginScope(

this ILogger logger,

string messageFormat,

params object?[] args)

{

if (logger == null)

{

throw new ArgumentNullException(nameof(logger));

}

return logger.BeginScope(new FormattedLogValues(messageFormat, args));

}

The Log extension method does the same thing.

<https://github.com/dotnet/runtime/blob/main/src/libraries/Microsoft.Extensions.Logging.Abstractions/src/LoggerExtensions.cs#L386-L394>

public static void Log(this ILogger logger, LogLevel logLevel, EventId eventId, Exception? exception, string? message, params object?[] args)

{

if (logger == null)

{

throw new ArgumentNullException(nameof(logger));

}

logger.Log(logLevel, eventId, new FormattedLogValues(message, args), exception, \_messageFormatter);

}

The \_messageFormatter? That's initialized assuming the FormattedLogValues internal class as well.

<https://github.com/dotnet/runtime/blob/main/src/libraries/Microsoft.Extensions.Logging.Abstractions/src/LoggerExtensions.cs#L13>

private static readonly Func<FormattedLogValues, Exception?, string> \_messageFormatter = MessageFormatter;

<https://github.com/dotnet/runtime/blob/main/src/libraries/Microsoft.Extensions.Logging.Abstractions/src/LoggerExtensions.cs#L425-L428>

private static string MessageFormatter(FormattedLogValues state, Exception? error)

{

return state.ToString();

}

It's only a couple methods that use it, so we can work around that, right? Nope, all the other extension methods in this class call the BeginScope and Log methods.

<https://github.com/dotnet/runtime/blob/main/src/libraries/Microsoft.Extensions.Logging.Abstractions/src/FormattedLogValues.cs#L16>

internal readonly struct FormattedLogValues : IReadOnlyList<KeyValuePair<string, object?>>

# Thoughts and Recommendations

I'm a huge fan of using the internal keyword to lockdown access for a class to the assembly it's defined in. Check out my article on [Assembly Encapsulation with Dependency Injection](https://www.danylkoweb.com/Blog/assembly-encapsulation-with-dependency-injection-LZ) to learn more.

I'm even fine with public interfaces defaulting an internal implementation. Check out my article on [Assembly Encapsulation with Dependency Injection – Unit Testing](https://www.danylkoweb.com/Blog/assembly-encapsulation-with-dependency-injection--unit-testing-SD) and [the MANY unit tests in my Tips.ApiMessage sample ToDo API](https://github.com/penblade/Tips/tree/master/Tips.ApiMessage/src).

What I struggle with is the concept of creating extension methods on a public interface with an internal implementation effectively making the interface pseudo-public.

**Recommended Standard**: If an extension method off a public interface needs to create a new instance of any object, then the class of that new instance should be public. I think this should be a standard when creating extension methods. I recommend changing FormattedLogValues from internal to public.

# Tips.Pipeline.Extensions.LoggerExtensions

Let's look at ILogger usage in this test app. The LogAction extension method begins the scope with a trace id, trace state, and the defined scope before performing the actual log. The trace id is a guid unique per request and the trace state includes some other tracking info. This gives the log context and makes it easy to follow a request through your code from beginning to end just dependent on how much you want to log.

<https://github.com/penblade/Tips/blob/e562ad1d0c032d34872ff5af14a17e2ea5fb26d0/Tips.ApiMessage/src/Pipeline/Extensions/LoggerExtensions.cs>

using System;

using Microsoft.Extensions.Logging;

namespace Tips.Pipeline.Extensions

{

public static class LoggerExtensions

{

public static void LogAction(this ILogger logger, string scope, Action method)

{

using (logger.BeginScopeWithApiTraceId())

using (logger.BeginScopeWithApiTraceStateString(scope))

using (logger.BeginScopeWithApiScope(scope))

{

method();

}

}

private static IDisposable BeginScopeWithApiTraceId(this ILogger logger) =>

logger.BeginScope("{Api.TraceId}", Tracking.TraceId);

private static IDisposable BeginScopeWithApiTraceStateString(this ILogger logger, string traceStateStringValue) =>

logger.BeginScope("{Api.TraceStateString}", Tracking.TraceStateString(traceStateStringValue));

private static IDisposable BeginScopeWithApiTraceStateStringEncoded(this ILogger logger, string traceStateStringValue) =>

logger.BeginScope("{Api.TraceStateString}", Tracking.TraceStateString(EncodeTraceStateStringValue(traceStateStringValue)));

private static IDisposable BeginScopeWithApiScope(this ILogger logger, string scope) =>

logger.BeginScope("{Api.Scope}", scope);

private static string EncodeTraceStateStringValue(string traceStateStringValue) =>

Convert.ToBase64String(System.Text.Encoding.Unicode.GetBytes(traceStateStringValue));

}

}

# Tips.Pipeline.Tracking

Tracking information for the logs. I'll discuss this more in future articles but included since its referenced.

<https://github.com/penblade/Tips/blob/e562ad1d0c032d34872ff5af14a17e2ea5fb26d0/Tips.ApiMessage/src/Pipeline/Tracking.cs>

using System.Diagnostics;

using System.Linq;

using System.Reflection;

namespace Tips.Pipeline

{

public static class Tracking

{

// https://www.w3.org/TR/trace-context/

// TraceParentStateString and TraceStateString values are not validated here.

// If these fields are critical for your tracking, then follow the detailed guidelines in the W3C trace-context.

public static string TraceParentId => Activity.Current?.ParentId;

public static string TraceId => Activity.Current?.Id;

public static string TraceParentStateString => Activity.Current?.TraceStateString;

public static string TraceStateString(string traceStateStringValue) => $"{ApplicationName}={traceStateStringValue}{ParseTraceParentStateString()}";

private static readonly string ApplicationName = Assembly.GetEntryAssembly()?.GetName().Name?.ToLower();

private static string ParseTraceParentStateString()

{

var state = TraceParentStateString?.Split(',').ToList().FirstOrDefault();

return !string.IsNullOrEmpty(state) ? $",{state}" : string.Empty;

}

}

}

# Tips.Pipeline.Logging.LoggingBehavior

LoggingBehavior is a [mediator pipeline (decorator) pattern as explained by Jimmy Bogard](https://lostechies.com/jimmybogard/2014/09/09/tackling-cross-cutting-concerns-with-a-mediator-pipeline/).

1. Controller calls the logging pipeline class (behavior)
   1. See the example [TodoItemsController](https://github.com/penblade/Tips/blob/master/Tips.ApiMessage/src/Api/Controllers/TodoItemsController.cs) for details, but it's not needed to understand the usage here
2. LoggingBehavior logs the request
3. LoggingBehavior calls the request handler to process the request and return a response
4. LoggingBehavior logs the response
5. LoggingBehavior returns the response to the controller
6. Controller returns the response with proper Http Status codes

using System.Text.Json;

using System.Threading.Tasks;

using Microsoft.Extensions.Logging;

using Tips.Pipeline.Extensions;

namespace Tips.Pipeline.Logging

{

internal class LoggingBehavior : IPipelineBehavior

{

private readonly ILogger<LoggingBehavior> \_logger;

public LoggingBehavior(ILogger<LoggingBehavior> logger) => \_logger = logger;

public async Task<TResponse> HandleAsync<TRequest, TResponse>(TRequest request, RequestHandlerDelegate<TResponse> nextAsync)

{

LogRequest(request);

var response = await nextAsync();

LogResponse(response);

return response;

}

private void LogRequest<TRequest>(TRequest request) =>

\_logger.LogAction("Request", () => \_logger.LogInformation("{Request}", JsonSerializer.Serialize(request)));

private void LogResponse<TResponse>(TResponse response) =>

\_logger.LogAction("Response", () => \_logger.LogInformation("{Response}", JsonSerializer.Serialize(response)));

}

# Tips.Pipeline.Tests.Logging.LoggingBehaviorTest

This unit test is fairly streamlined going through the paces to verify the logic for logging is in place.

using System.Text.Json;

using System.Threading.Tasks;

using Microsoft.Extensions.Logging;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using Moq;

using Tips.Support.Tests;

using Tips.Pipeline.Logging;

namespace Tips.Pipeline.Tests.Logging

{

[TestClass]

public class LoggingBehaviorTest

{

[TestMethod]

public async Task HandleAsyncTest()

{

const string requestString = "Request";

const string responseString = "Response";

var fakeRequest = new FakeRequest { FakeIntProperty = 42, FakeStringProperty = "Test String" };

var serializedFakeRequest = JsonSerializer.Serialize(fakeRequest);

var fakeResponse = new FakeResponse { FakeBoolProperty = true, FakeDecimalProperty = 3.6M };

var serializedFakeResponse = JsonSerializer.Serialize(fakeResponse);

var mockLogger = new Mock<ILogger<LoggingBehavior>>();

//var loggingBehavior = new LoggingBehavior(fakeLogger);

var loggingBehavior = new LoggingBehavior(mockLogger.Object);

var mockRequestHandlerDelegate = new Mock<RequestHandlerDelegate<FakeResponse>>();

mockRequestHandlerDelegate.Setup(x => x.Invoke()).ReturnsAsync(fakeResponse);

var actualResponse = await loggingBehavior.HandleAsync(fakeRequest, mockRequestHandlerDelegate.Object);

mockLogger.VerifyBeginScope(requestString);

mockLogger.VerifyBeginScope(responseString);

mockLogger.VerifyLog(LogLevel.Information, serializedFakeRequest, requestString);

mockLogger.VerifyLog(LogLevel.Information, serializedFakeResponse, responseString);

Assert.AreSame(fakeResponse, actualResponse);

}

public class FakeRequest

{

public int FakeIntProperty { get; set; }

public string FakeStringProperty { get; set; }

}

public class FakeResponse

{

public bool FakeBoolProperty { get; set; }

public decimal FakeDecimalProperty { get; set; }

}

}

}

# Tips.Support.Tests.MockLoggerVerifyLogExtensions

We want to be as exact as possible to ensure the tests pass for the right reasons. Thankfully, [Christian Findlay](https://christianfindlay.com) enlightened us with the pattern below using It.IsAnyType with the Moq framework. For each use of It.IsAnyType below, we would prefer to use FormattedLogValues, but we can't because its internal and not accessible. This will just have to be close enough, which is still pretty good, without making an actual integration test.

using System;

using System.Collections.Generic;

using System.Linq;

using Microsoft.Extensions.Logging;

using Moq;

namespace Tips.Support.Tests

{

public static class MockLoggerVerifyLogExtensions

{

public static void VerifyBeginScope<TCategory>(this Mock<ILogger<TCategory>> mockLogger, string state) =>

mockLogger.Verify(logger => logger.BeginScope(It.Is<object>(request => request.ToString() == state)), Times.Once);

public static void VerifyLog<TCategory>(this Mock<ILogger<TCategory>> mockLogger, LogLevel logLevel, object message, string key) =>

mockLogger.Verify(logger =>

logger.Log(

logLevel,

It.IsAny<EventId>(),

It.Is<It.IsAnyType>((state, t) => CheckValue(state, message, key)),

null,

(Func<It.IsAnyType, Exception, string>)It.IsAny<object>())

, Times.Once);

public static void VerifyLog<TCategory>(this Mock<ILogger<TCategory>> mockLogger, LogLevel logLevel, Exception exception, string key) =>

mockLogger.Verify(logger =>

logger.Log(

logLevel,

It.IsAny<EventId>(),

It.Is<It.IsAnyType>((state, t) => CheckValue(state, null, key)),

It.Is<Exception>(actualException => CheckException(exception, actualException)),

(Func<It.IsAnyType, Exception, string>)It.IsAny<object>())

, Times.Once);

/// <summary>

/// Checks that a given key exists in the given collection, and that the value matches

/// \*\*\* Updated to get FirstOrDefault. \*\*\*

/// Credit: https://christianfindlay.com/2020/07/03/ilogger/

/// Code: https://github.com/MelbourneDeveloper/Samples/blob/650ba4bc6cba631da651c2c6732bce337e6a7d8e/ILoggerSamples/ILoggerSamples/ILoggerTests.cs#L44

/// </summary>

private static bool CheckValue(object state, object expectedValue, string key)

{

var keyValuePairList = (IReadOnlyList<KeyValuePair<string, object>>)state;

var actualValue = keyValuePairList.FirstOrDefault(kvp =>

string.Compare(kvp.Key, key, StringComparison.Ordinal) == 0).Value;

// Support null values.

if (expectedValue == null && actualValue == null) return true;

return expectedValue?.Equals(actualValue) ?? false;

}

/// <summary>

/// Checks that a given exception was called.

/// </summary>

private static bool CheckException(Exception expectedException, Exception actualException) =>

expectedException?.GetType() == actualException?.GetType()

&& expectedException?.Message == actualException?.Message;

}

}

# References

1. [GIT: Penblade/Tips/Tips.ApiMessage.src](https://github.com/penblade/Tips/tree/e562ad1d0c032d34872ff5af14a17e2ea5fb26d0/Tips.ApiMessage/src)
   1. This is the sample Todo api used as reference
   2. Includes ILogger extension methods
   3. Includes Moq extension methods for verifying ILogger was called
   4. Includes plenty of tests for ILogger unit test variations

# Summary

I showed how ILogger extension methods, which are super useful, use internal class FormattedLogValues. I demonstrated how this prevents our unit tests from being as exact as possible. I provided ILogger extension methods with tracking info. I provided Moq extension methods to assist with verifying ILogger log methods were called. I included sample code for verifying calls to ILogger.

*Do you agree with the recommendation to not use internal classes in public extension methods? Have you run into this ILogger concern in your unit tests? Do you find the ILogger or Moq extension methods useful?*