



EFFECTIVE LOGGING FOR OPTIMAL OBSERVABILITY

A guide to logging practices that produce highly observable systems

Abstract

This document offers a practical guide, for software engineers, to logging practices that produce highly observable systems. It defines key terms and concepts related to system observability and logging, including the Four Golden Signals, and down-stream concerns such as service level indicators, service level objectives, and service level agreements. By following the guidelines presented in this document, software teams can produce systems that balance the need for observability, code maintainability, and system performance.

Lauter, Mark
Mark.Lauter@insight.com

Contents

Introduction	2
Key Terms and Concepts	2
Observability	2
The Four Golden Signals.....	2
Latency	2
Traffic Volume	2
Error Rate	2
Saturation.....	2
Service Level Indicators.....	2
Service Level Objectives.....	3
Service Level Agreements	3
Logging	3
Log.....	3
Telemetry	3
Monitoring	3
Incident Detection and Alerting.....	3
Incident Resolution and Troubleshooting	4
Incident Prevention and Planning	4
What to Log.....	4
Event Metadata.....	5
Request Metadata.....	5
Operational Context.....	6
System Faults: Exceptions and Warnings	6
Levels.....	6
How to Log	7
ILogger<TCategoryName>.....	7
Serilog	10
XUnit	12
XUnitTestOutputLoggerProvider	12
Serilog	14
References.....	18

Introduction

The purpose of this document is to provide software developers with a guide to logging practices that produce highly observable systems. The document contains examples in C# using the Serilog implementation of the `Microsoft.Extensions.Logging.ILogger` interface.

Key Terms and Concepts

A mutual understanding of key terms and concepts is essential for effective communication and collaboration among team members. This section establishes a vocabulary of key terms and concepts related to system observability and logging for software developers, operations teams, such as SRE, and other stakeholders.

Observability

Observability is the ease with which the behavior and performance of a system may be monitored and measured for the purposes of maintenance, troubleshooting and continuous improvement. A system with optimal observability emits telemetry that allows monitoring and measurement of the system reliability engineering key performance indicators, known as “The Four Golden Signals,” without unduly affecting system performance nor complexity.

The Four Golden Signals

The Four Golden Signals measure the health of a system. Each signal represents an aspect of system behavior:

Latency

Latency is the time it takes for a system to respond to, or process, a request.

Traffic Volume

Traffic volume is the ratio of requests received by a system over time.

Error Rate

Error rate is the ratio of failed requests over successful requests or the ratio of system faults, or exceptions, over time.

Saturation

Saturation is the load on infrastructure, such as memory consumption, CPU usage, or network congestion.

Service Level Indicators

The signals are collections metrics known as Service Level Indicators (SLI). An SLI is a metric which expresses key performance information related to a system. SLIs are expressed as simple ratios. For example, the SLI describing the latency of a system could be defined as the number of requests served within a desired timespan over the total number of requests. Consider a typical REST API that received 500,000 requests in the last hour. Of those 500k requests, 400k were served within the expected duration of 10 milliseconds. The calculation for the latency SLI is 400 requests served within 10 milliseconds divided by 500 total requests = 0.8. Our SLI for the latency of our REST API is 0.8 or 80%.

Service Level Objectives

A Service Level Objective (SLO) is a specific SLI target or goal. Consider the SLI example above. If our SLO, or target SLI, for latency is 0.9 and the measured, or actual, latency is 0.8, then the SLI is 11% below target. In this case 0.9 is the SLO and 0.8 is the SLI. An SLO may be defined as a fixed ratio or a range, such as standard deviation. SLOs are selected based on user expectations, business requirements, resource and cost constraints, industry standards, and historical data.

Service Level Agreements

A Service Level Agreement is a formal commitment by a service provider to consumers of the service that defines:

- service description
- service limitations
- service Level Objectives
- responsibilities of the service provider
- responsibilities of the service users
- monitoring and reporting

Logging

Logging is the emission and recording of events related to the behavior of a software system. Events may be related to system faults, performance metrics, user activity, system access, process flow, or any other data that offers insight into the behavior of the system for the purpose of maintenance, planning, cost estimation, incident detection, incident resolution, and incident prevention. A stream of these events is known as telemetry. Effective logging practices balance the need for observability, code maintainability, and system performance.

Log

A log is a semi-durable record of the events emitted by a system. Logs are typically structured as append-only time-series data stores that include event metadata such as timestamp, description, severity or level, and any relevant context. Logs are captured by event sinks. An event sink might direct log events directly to stdout, to a file, or via API to an external APM service such as CloudWatch, Splunk or Application Insights.

Telemetry

Telemetry is the practice of collecting, transmitting, and analyzing data related to the behavior and performance of a remote system. The goal of telemetry is to provide insights that can be used for informed decision making regarding the future of the system, detection, resolution, and prevention of incidents. Good telemetry leads to high observability.

Monitoring

Monitoring is the transformation of system logs into meaningful projections such as charts, status reports, dashboards, and alert notifications.

Incident Detection and Alerting

Responsible operation of a software system requires proactive detection and resolution of incidents. An incident is triggered when any SLO is breached. For example, when the acceptable latency for an API is

exceeded over a certain number of requests then a breach of the latency SLO occurs, and this breach is raised as an incident. Relying on users to report such issues negatively impacts the user experience, delays issue resolution, and may result in violation of SLAs.

With good telemetry in place, system issues, such as system faults, increased latency, over-saturation, and changes in traffic volume, can be detected by an application performance monitoring (APM) system. Issues identified by the APM can be forwarded to an alert management system (AMS) which forwards the notification to a human agent who can then intervene before the issue escalates to user impact. Specific guidance for incident detection, APM and AMS are outside the scope of this document.

Incident Resolution and Troubleshooting

To quickly resolve incidents, operation team members require enough context to effectively troubleshoot. For any software system, this includes relevant information such as event source, error messages, and stack traces. For distributed systems, the context should include relevant infrastructure details, such as server id and operations region, as well as correlation, or trace, ids that make it possible to trace requests across multiple services. Developers will also be interested in more detailed context such as class name, function name, and relevant system state. Including context in the system logs makes it easier to pinpoint the root cause of the issue. Uncovering the root cause is often the first step in resolving an incident, so the faster an operator can id the root cause, the faster a solution can be devised, implemented, and deployed by the team. It is the software developer's responsibility to include this context in the system's telemetry.

Incident Prevention and Planning

Incident prevention and planning is the process of proactively identifying and mitigating potential risks and vulnerabilities in a system and establishing contingencies to minimize the impact of incidents that cannot be prevented. Effective incident prevention and planning practices reduce the likelihood and severity of incidents and improve the overall health and reliability of the system.

What to Log

The key terms and concepts section introduces important goals of system telemetry:

- detection of incidents, errors, or faults
- troubleshooting and resolution of problems
- identification of trends requiring mitigation
- establishment of service level objectives
- establishment of service level agreements

Good logs support these goals by focusing on the Four Golden Signals:

- Latency: Log performance metrics such as API response times, end-to-end workflow duration, method, or algorithm duration, third-party, or external, API response times, SQL query duration. Including a timestamp on all log events makes it possible to aggregate latency through the APM.
- Traffic Volume: Log in-bound API requests, out-bound API requests, method calls, user activity, request payload size, and important business events such as orders placed, or payments received. Volume is typically aggregated by the APM, but for events that occur too often to be logged individually, metrics must be aggregated by the software system and logged periodically.

- **Error Rate:** Log exceptions along with any relevant context that will be helpful when diagnosing, or troubleshooting, incidents. An exception is any unexpected result returned by a process, function, or request. Add context by including exception meta data such as exception type, exception message, stack trace, system state or other operational context, and always include the full exception tree by iterating through the inner exceptions.
- **Saturation:** Log infrastructure usage metrics, such as CPU usage, memory consumption, and network bandwidth in-use. In cloud environments, such as AWS ECS and AWS Lambda functions, saturation is logged automatically.

Event Metadata

Every log entry should have a common set of meta data that includes, but may not be limited to:

- timestamp
- log level
- event source
- region
- server ID
- applicable environment variables
- run-time options
- system version
- session ID
- correlation, or trace, ID
- user ID, if applicable
- payload size, item count, or other volume related metric, if applicable

Request Metadata

Every REST API request should include the following information:

- request method (GET, POST, etc.)
- request endpoint URL
- request event ID
- request headers
- response status code
- response headers
- request duration
- request source
 - user agent
 - client IP
 - user ID
 - user role
 - user authentication method (APIKey, token, etc.)

Sometimes requests are handled by sub-system types other than REST APIs. For example, requests are routed to AWS Lambdas through many different trigger sources including HTTP, SQS, DynamoDB, etc. The basic information to be included slightly different than that included for a REST API request.

- trigger type (SQS, DynamoDB, S3, etc.)
- trigger source (name of the specific queue, table, bucket, etc.)
- trigger source event ID
- count of event messages reported by the trigger event
- message ID of each message processed
- time to process each message
- time to process all messages in aggregate
- success or failure per message

Operational Context

Operational context includes information related to a particular event. Supplying context makes it possible for someone reading a system log to better understand the circumstances of the event and to gain insight into the behavior of the system. Obviously, context is situationally dependent, and some context is already provided within the Meta Data, Request, and Exceptions sections, but here are some guidelines to follow:

- include unique, meaningful log messages
- include method names and line numbers, when appropriate
- include IDs and other relevant properties of objects within the scope of the event
- include business specific context

System Faults: Exceptions and Warnings

Context is especially important to someone troubleshooting an incident. Always include the minimum when logging exceptions:

- exception type name
- exception message
- exception source
- exception data
- stack trace
- full exception tree (iterate through the inner exceptions)
- all exceptions within an aggregate exception
- any data related to the cause of the exception

Always compose custom exceptions for your libraries and never throw `System.Exception`. Error rates for a specific service, library or sub-system cannot be correctly calculated when all the exceptions are of the same type.

Levels

Log levels are a way to categorize and filter events based on their severity. It is important for developers to specify the appropriate level when writing entries because the level meta data element provides additional context for anyone analyzing the log.

There are seven log levels. From lowest severity to highest, they are:

- Trace – Events with the most detailed information and may include sensitive data. Useful default level only for local debugging.
- Debug – Events for debugging and development. Useful default level for local debugging and for development or integration or other low volume test environments. Not for use in production due to high volume.
- Information – Events that track the general flow of the system and are a primary source of data for SLIs. Useful default level for production environments.
- Warning – Events that track errors or other unexpected events that will not cause the system to abort or the process to fail. Useful default level for third party libraries.
- Error – Events that track unhandled exceptions within the context of the current operation.
- Fatal – Events that track severe application-wide faults that will cause the system to abort, such as out of memory, missing environment variable, etc.
- None – No log messages will be written.

How to Log

In .Net logging follows the same DI pattern as any other service so it will require registration of a specific logger implementation with the IoC container and injection of the ILogger interface into classes requiring access to the logging service. This section will cover both IoC container registration using the standard Microsoft IHost interface and the injection and standard use of the ILogger interface.

IoC and the specifics of .Net Dependency Injection (DI) are beyond the scope of this document, but here is a quick refresher. IoC is implemented in three parts:

- inject services into a dependent class via the dependent class's constructor
- register services with an IoC container such as IServiceCollection
- resolve and construct services at run time through a service provider such as ServiceProvider

You can learn more about DI here: [Dependency injection - .NET | Microsoft Learn](#)

The most important thing to remember is that libraries defer the selection of implementation to the runtime executable. In other words, always program to an interface, not to a specific concrete implementation. The concrete implementation will be selected by the service configuration in the runtime project. This means your library projects do not need to, and never should, install Serilog packages.

All sample code included in this section is available here:

[marklauter/logsample: log samples \(github.com\)](https://github.com/marklauter/logsamples)

ILogger<TCategoryName>

All logging in .Net is performed through the ILogger<TCategoryName> interface. For detailed information regarding ILogger<TCategoryName> read the .Net documentation here: [ILogger<TCategoryName> Interface \(Microsoft.Extensions.Logging\) | Microsoft Learn](#)

To get started using `ILogger<TCategoryName>` install the following package:

- `Microsoft.Extensions.Logging.Abstractions`

Next, to enable logging in your library class simply inject `ILogger<TCategoryName>` into your class and supply the class as the type argument. You can then write log entries with an appropriate log level.

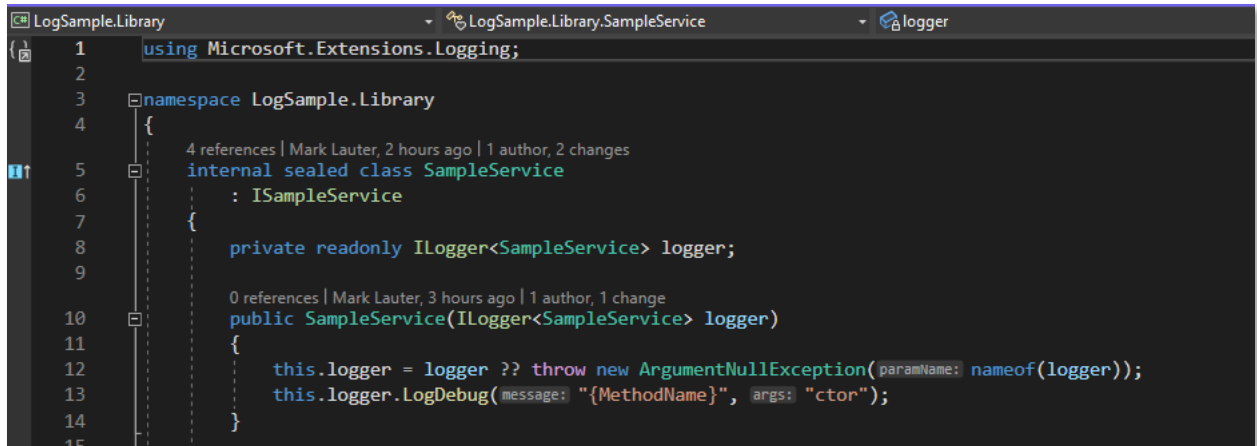


Figure 1 Injection of `ILogger`

`ILogger<TCategoryName>` exposes extension methods that make it easier to write log entries. Each method obviously targets a log level:

- `LogTrace`
- `LogDebug`
- `LogInformation`
- `LogWarning`
- `LogError`
- `LogCritical`

Each of these methods accepts a message template and a params array. The message argument specifies a message template which should be a string constant. **Do not pass interpolated strings.** First, string constants are memory optimized by the .Net compiler. Second logger implementations typically cache the message templates as they are expected to be unique and unchanging. Passing interpolated strings will create a massive load on the GC and severely degrade system performance. Additionally, it will obfuscate the meta data that should be included in the message template. To learn more, checkout this video by Nick Chapsas: [Stop using String Interpolation when Logging in .NET - YouTube](#)

Values are passed to the log by including a named element in the template. Element names are always capitalized and surrounded by curly braces. For example, `UserName` is a named element in the following template: `"My name is {UserName}"`

The argument containing the value for UserName is passed as an argument to the Log method like this:

```
logger.LogDebug("My name is {UserName}", user.Name);
```

This sample code illustrates the right and wrong ways to write a log event.

```
this.logger.LogTrace(message: $"This is a {LogLevel.Trace} message.", args: LogLevel.Trace); // WRONG
this.logger.LogTrace(message: "This is a {Level} message.", args: LogLevel.Trace); // RIGHT
```

Figure 2 Don't use string interpolation example

Code hint ellipsis shows the issue.

```
this.logger.LogTrace(message: $"This is a {LogLevel.Trace} message.", args: LogLevel.Trace); // WRONG
this.logger.LogTrace(message: "This is a {Level} message.", args: LogLevel.Trace); // RIGHT
```

class System.String
Represents text as a sequence of UTF-16 code units.

CA2254: The logging message template should not vary between calls to 'LoggerExtensions.LogTrace(Logger, string?, params object?[])'

Figure 3 Don't use string interpolation example

Using a non-constant string is also unacceptable.

```
var string? messageTemplate = "This is a {Level} message.";
this.logger.LogTrace(message: messageTemplate, args: LogLevel.Trace); // WRONG
this.logger.LogDebug(message: messageTemplate, args: LogLevel.Debug); // WRONG

this.logger.LogTrace(message: "This is a {Level} message.", args: LogLevel.Trace); // RIGHT
this.logger.LogDebug(message: "This is a {Level} message.", args: LogLevel.Debug); // RIGHT
```

Figure 4 Don't use string interpolation example

Compiler generates the same hint as before.

```
var string? messageTemplate = "This is a {Level} message.";
this.logger.LogTrace(message: messageTemplate, args: LogLevel.Trace); // WRONG
this.logger.LogDebug(message: messageTemplate, args: LogLevel.Debug); // WRONG

this.logger.LogTrace(message: "This is a {Level} message.", args: LogLevel.Trace); // RIGHT
this.logger.LogDebug(message: "This is a {Level} message.", args: LogLevel.Debug); // RIGHT
```

(local variable) string? messageTemplate
'messageTemplate' is not null here.

CA2254: The logging message template should not vary between calls to 'LoggerExtensions.LogDebug(Logger, string?, params object?[])'

Figure 5 Don't use string interpolation example

Complex data types can be expressed in the log entry by including the @ symbol before the element name in the template. They will typically be projected in the output as JSON, but this is dependent upon the logger implementation and exact projection format will vary.

```
this.logger.LogInformation(message: "info: {@Data}", args: data);
```

Figure 6 Use of ILogger for writing log events

When logging exceptions use LogWarning, LogError, or LogCritical depending on the severity of the fault. Always pass the exception, if the fault is the result of a catch clause, as the first argument. This way exception meta data can be expressed in the log entry.

```
26 3 references | Mark Lauter, 2 hours ago | 1 author, 2 changes
27 public void Fail(SampleData data)
28 {
29     using var IDisposable? logScope = this.logger.BeginScope(messageFormat: "{MethodName}", args: nameof(Fail));
30     try
31     {
32         this.logger.LogInformation(message: "{@Data}", args: data);
33
34         throw new SampleOuterException(
35             message: $"fail: {nameof(SampleOuterException)}",
36             innerException: new SampleInnerException(message: $"fail: {nameof(SampleInnerException)}")
37             {
38                 Data = { { "SampleData", data } }
39             });
40     }
41     catch (SampleOuterException ex)
42     {
43         this.logger.LogError(exception: ex, message: "can't do the thing with this data: {@Data}", args: data);
44     }
45     catch (Exception ex)
46     {
47         this.logger.LogCritical(exception: ex, message: "unexpected exception while doing the thing. system abort time.");
48         throw;
49     }
50 }
```

Figure 7 Use of ILogger to write exception log events

Serilog

In 2023 Serilog is the only acceptable concrete implementation of the `ILogger<TCategoryName>` interface. Log4Net is deprecated due to security issues and the built in Microsoft logger implementation has a limited feature set.

Serilog uses a pluggable structure that offers complete control over the structured output and targeting of specific sinks. This guide is a primer. For advanced use cases read the official Serilog documentation available at GitHub: [Serilog \(github.com\)](https://github.com/serilog/serilog)

Serilog automatically includes a lot of the recommended meta data included in the “What to Log” section of this document. For example, by default it includes, timestamp, event source (which comes from `TCategoryName`) and log level. By including the Exceptions package and simple configuration tweak it will also output full exception data including traversal of the inner exception tree.

To get started with Serilog, install the following packages:

- Serilog
- Serilog.Exceptions
- Serilog.Formatting.Compact

To use the `IHost` extensions for ASP.NET Core or console applications, install:

- Serilog.Extensions.Hosting

To add Serilog as the default logger builder in an XUnit project, install:

- Serilog.Extensions.Logging

To load custom settings from configuration, install:

- Serilog.Settings.Configuration

To include request logging middleware, install:

- Serilog.AspNetCore

There many are useful sinks, but the most common require these packages:

- Serilog.Sinks.Console
- Serilog.Sinks.Debug
- Serilog.Sinks.File
- Serilog.Sinks.XUnit

Your project packages item group will look something like this:

```
<ItemGroup>
  <PackageReference Include="Microsoft.Extensions.Hosting" Version="7.0.1" />

  <PackageReference Include="Serilog" Version="2.12.0" />
  <PackageReference Include="Serilog.Exceptions" Version="8.4.0" />
  <PackageReference Include="Serilog.Extensions.Hosting" Version="5.0.1" />
  <PackageReference Include="Serilog.Formatting.Compact" Version="1.1.0" />
  <PackageReference Include="Serilog.Settings.Configuration" Version="3.4.0" />
  <PackageReference Include="Serilog.Sinks.Console" Version="4.1.0" />
  <PackageReference Include="Serilog.Sinks.Debug" Version="2.0.0" />

  <PackageReference Include="IDisposableAnalyzers" Version="4.0.6">
    <PrivateAssets>all</PrivateAssets>
    <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
  </PackageReference>
</ItemGroup>
```

Figure 8 Typical Serilog package references

A typical service configuration using the IHost extensions will look something like this:

```
1 using LogSample.Library;
2 using Microsoft.Extensions.DependencyInjection;
3 using Microsoft.Extensions.Hosting;
4 using Serilog;
5 using Serilog.Exceptions;
6 using Serilog.Formatting.Compact;
7 using System.Reflection;
8
9 using var IHost? host = Host.CreateDefaultBuilder(args)
10     .ConfigureServices((configureDelegate: (HostBuilderContext hostContext, IServiceCollection services) => _ = services
11         .AddSampleService()))
12     .UseSerilog((configureLogger: (HostBuilderContext builderContext, IServiceProvider services, LoggerConfiguration configuration) => _ = configuration
13         .MinimumLevel.Verbose()
14         .Enrich.FromLogContext()
15         .Enrich.WithExceptionDetails()
16         .WriteTo.Console(formatter: new CompactJsonFormatter())
17         #if DEBUG
18         .WriteTo.Debug(formatter: new CompactJsonFormatter())
19         #endif
20         .ReadFrom.Configuration(configuration: builderContext.Configuration)
21         .Enrich.WithProperty(name: "Application.Name", value: "LogSample.Console")
22         .Enrich.WithProperty(name: "Application.Version", value: Assembly.GetExecutingAssembly().GetName().Version?.ToString() ?? "local-debug"))
23     .Build();
```

Figure 9 IHost.UseSerilog sample

It is worth noting that everything specified in the UseSerilog lambda can also be specified in configuration. The specifics of .Net configuration and how to implement configuration for the many project types is beyond the scope of this document.

To include request logging middleware for a web application, use the WebApplication extension method UseSerilogRequestLogging. Remember that the execution of the middleware stack is ordered by the order in which the middleware is added to the WebApplication instance. So, if you want to log the requests before other middleware runs, then place the call to UseSerilogRequestLogging at the top of the middleware chain.

XUnit

Logging in unit tests is a special case. Well written unit tests are black-box tests. They set up precondition state, execute a process, and then assert on post-execution state mutations. Knowledge of the internals of the process violates the principle of encapsulation. For this reason, log output typically is ignored by unit tests, but there may be a situation where the developer wants to test the correctness of log output. An argument can certainly be made that log output represents a post-execution state mutation. For such cases, the XUnit Dependency Injection package includes ITestOutputHelper. Using either a custom Serilog sink, or the custom XUnitTestOutputLoggerProvider makes it possible to redirect output to an instance of ITestOutputHelper. The results of the logging operations can then be read from ITestOutputHelper. This section covers both Serilog and XUnitTestOutputLoggerProvider options.

XUnitTestOutputLoggerProvider

XUnitTestOutputLoggerProvider is the simplest solution, but the structured output will not be an exact match to production logs generated by Serilog.

The packages required to capture simple logging in XUnit are:

- Xunit.DependencyInjection
- Xunit.DependencyInjection.Logging

The setup required is simple. Just AddLogging to the service collection and add the Configure method with a call to ILoggerFactory's AddProvider method. Now the unit test project is ready.

```

LogSample.Tests.XUnit
LogSample.Tests.XUnit.Startup
ConfigureServices(IServiceCollection services)
1 using Microsoft.Extensions.DependencyInjection;
2 using Microsoft.Extensions.Logging;
3 using Xunit.DependencyInjection;
4 using Xunit.DependencyInjection.Logging;
5
6 #pragma warning disable CA1822 // Mark members as static
7 #pragma warning disable IDISP004 // Don't ignore created IDisposable
8
9 namespace LogSample.Tests.XUnit
10 {
11     public sealed class Startup
12     {
13         public void ConfigureServices(IServiceCollection services)
14         {
15             _ = services
16                 .AddLogging()
17                 .AddSampleService();
18         }
19
20         public void Configure(ILoggerFactory loggerFactory, ITestOutputHelperAccessor accessor)
21         {
22             loggerFactory.AddProvider(
23                 provider: new XunitTestOutputLoggerProvider(
24                     accessor,
25                     filter: (string source, LogLevel ll) => ll >= LogLevel.Trace));
26         }
27     }
28 }
29
30 #pragma warning restore IDISP004 // Don't ignore created IDisposable
31 #pragma warning restore CA1822 // Mark members as static
32

```

Figure 10 Xunit logger provider setup

Inject ITestOutputHelper and your service to be tested into the unit test class.

```

1 using Xunit.Abstractions;
2
3 namespace LogSample.Tests.XUnit
4 {
5     public sealed class SampleServiceTests
6     {
7         private readonly ITestOutputHelper output;
8         private readonly ISampleService sampleService;
9
10        public SampleServiceTests(
11            ISampleService sampleService,
12            ITestOutputHelper output)
13        {
14            this.sampleService = sampleService ?? throw new ArgumentNullException(nameof(sampleService));
15            this.output = output ?? throw new ArgumentNullException(nameof(output));
16        }
17    }
18 }

```

Figure 11 ITestOutputHelper injection

Now logger output can be tested.

```
18 [Fact]
19 | 1 reference | 1/1 passing | Mark Lauter, 23 hours ago | 1 author, 2 changes
20 public void Succeed_Succeeds()
21 {
22     var SampleData sampleData = new SampleData(
23         Id: Guid.NewGuid(),
24         Name: nameof(Succeed_Succeeds),
25         TimestampUtc: DateTime.UtcNow);
26     this.sampleService.Succeed(data: sampleData);
27
28     var string? stdout = ((Xunit.Sdk.TestOutputHelper)this.output).Output;
29     Assert.Contains(expectedsubstring: "SampleData", actualString: stdout);
30     Assert.Contains(expectedsubstring: sampleData.Id.ToString(), actualString: stdout);
31 }
```

Figure 12 using `ITestOutputHelper` to test log output example one

Here is the output generated by the unit test.

```
Test Detail Summary
✓ LogSample.Tests.XUnit.SampleServiceTests.Succeed_Succeeds
Source: SampleServiceTests.cs line 19
Duration: 26 ms
Standard Output:
info: LogSample.Library.SampleService[0]
      info: SampleData { Id = fe63f9ea-8882-47ec-95b8-57406f0885e6, Name = Succeed_Succeeds, TimestampUtc = 10 May 23 16:36:21 }
```

Figure 13 Unit test output generated by `XUnitTestOutputLoggerProvider`

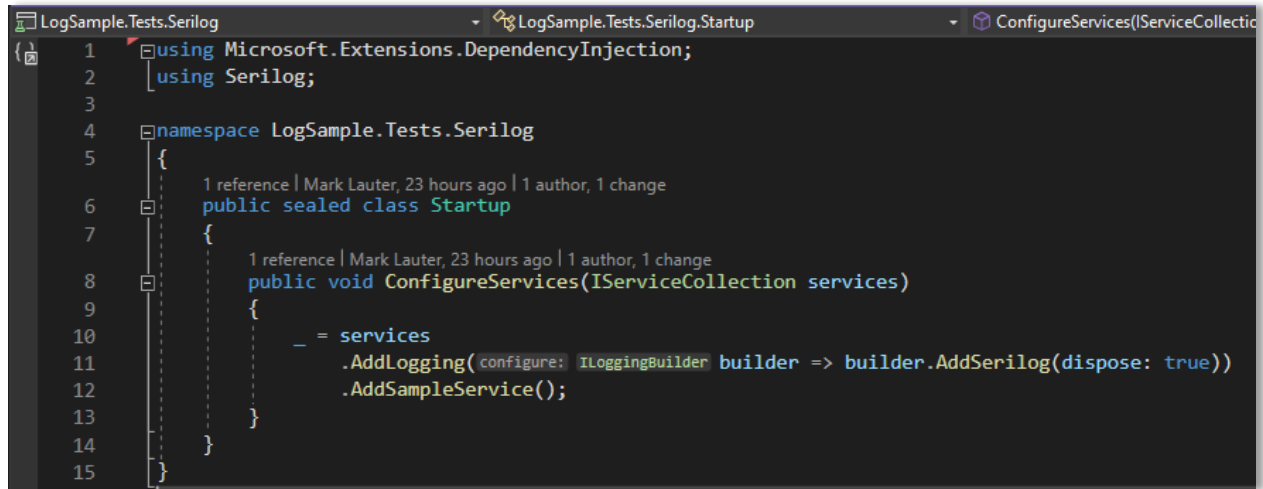
Serilog

Serilog is a slightly more advanced option for generating structured log output in unit tests. While it may add some complexity, it may be worth it if the tests require precise modeling the production structured log output.

Using Serilog with XUnit requires these packages:

- Serilog
- Serilog.Exceptions
- Serilog.Extensions.Logging
- Serilog.Formatting.Compact (only if you use compact formatting in production)
- Serilog.Sinks.XUnit

Setup is simple, just specify Serilog as the default log builder when calling AddLogging to add logging services to the service collection.



```
1 using Microsoft.Extensions.DependencyInjection;
2 using Serilog;
3
4 namespace LogSample.Tests.Serilog
5 {
6     1 reference | Mark Lauter, 23 hours ago | 1 author, 1 change
7     public sealed class Startup
8     {
9         1 reference | Mark Lauter, 23 hours ago | 1 author, 1 change
10        public void ConfigureServices(IServiceCollection services)
11        {
12            _ = services
13                .AddLogging(configure: iLoggingBuilder builder => builder.AddSerilog(dispose: true))
14                .AddSampleService();
15        }
16    }
17 }
```

Figure 14 XUnit loggerBuilder setup with Serilog

Extra complexity is introduced with the logger injection. Start by injecting the service provider instead of the test subject interface, then setup Serilog and target `ITestOutputHelper` with the `Serilog.XUnit.Sink.WriteTo` extension, and finally use the service provider to retrieve an instance of the test subject interface.

```
1 using Microsoft.Extensions.DependencyInjection;
2 using Serilog;
3 using Serilog.Exceptions;
4 using Serilog.Formatting.Compact;
5 using Xunit.Abstractions;
6
7 namespace LogSample.Tests.Serilog
8 {
9     1 reference | Mark Lauter, 23 hours ago | 1 author, 1 change
10     public sealed class SampleServiceTests
11     {
12         private readonly ITestOutputHelper output;
13         private readonly ISampleService sampleService;
14
15         0 references | Mark Lauter, 23 hours ago | 1 author, 1 change
16         public SampleServiceTests(
17             IServiceProvider serviceProvider,
18             ITestOutputHelper output)
19         {
20             if (serviceProvider is null)
21             {
22                 throw new ArgumentNullException(paramName: nameof(serviceProvider));
23             }
24
25             this.output = output ?? throw new ArgumentNullException(paramName: nameof(output));
26             Log.Logger = new LoggerConfiguration()
27                 .MinimumLevel.Verbose()
28                 .Enrich.FromLogContext()
29                 .Enrich.WithExceptionDetails()
30                 .WriteTo.TestOutput(testOutputHelper: output, formatter: new CompactJsonFormatter())
31                 .CreateLogger();
32
33             this.sampleService = serviceProvider.GetRequiredService<ISampleService>();
34         }
35     }
36 }
```

Figure 15 Serilog injection into unit test

The actual test is identical to the previous example; however, the output is structured differently.

```
[Fact]
✓ | 1 reference | ✓ 1/1 passing | Mark Lauter, 23 hours ago | 1 author, 2 changes
public void Succeed_Succeeds()
{
    var SampleData sampleData = new SampleData(
        Id: Guid.NewGuid(),
        Name: nameof(Succeed_Succeeds),
        TimestampUtc: DateTime.UtcNow);
    this.sampleService.Succeed(data: sampleData);

    var string? stdout = ((Xunit.Sdk.TestOutputHelper)this.output).Output;
    Assert.Contains(expectedSubstring: "SampleData", actualString: stdout);
    Assert.Contains(expectedSubstring: sampleData.Id.ToString(), actualString: stdout);
}
```

Figure 16 using *ITestOutputHelper* to test log output example two

Here is the output generated by the unit test.

```
Test Detail Summary
✓ LogSample.Tests.Serilog.SampleServiceTests.Succeed_Succeeds
  Source: SampleServiceTests.cs line 35
  Duration: 54 ms
  Standard Output:
    {"@t":"2023-05-10T16:36:21.5769177Z","@mt":"{MethodName}","@l":"Debug","MethodName":"ctor","SourceContext":"LogSample.Libr
    {"@t":"2023-05-10T16:36:21.5915268Z","@mt":"This is a {Level} message.","@l":"Verbose","Level":"Trace","SourceContext":"Lo
    {"@t":"2023-05-10T16:36:21.5950930Z","@mt":"This is a {Level} message.","@l":"Debug","Level":"Debug","SourceContext":"LogS
    {"@t":"2023-05-10T16:36:21.5986810Z","@mt":"info: {@Data}","Data":{"Id":"27696366-bd79-4ab6-a623-771bd1b7aa64","Name":"Suc
```

Figure 17 Unit test output generated by *Serilog Compact Json Formatter*

References

- [Logging in C# - .NET | Microsoft Learn](#)
- [Serilog Best Practices - Ben Foster](#)
- [Logging and reporting decision guide - Cloud Adoption Framework | Microsoft Learn](#)
- [Serilog — simple .NET logging with fully-structured events](#)
- [serilog/serilog: Simple .NET logging with fully-structured events \(github.com\)](#)
- [Serilog Tutorial for .NET Logging: 16 Best Practices and Tips \(stackify.com\)](#)
- [.NET Logging Basics - The Ultimate Guide To Logging \(loggly.com\)](#)
- [Observability Primer | OpenTelemetry](#)
- [Distributed Monitoring 101: the “Four Golden Signals” | by Vincent Gilles | ForePaaS | Medium](#)
- [The 4 Golden Signals, and how to put them into practice | TechTarget](#)
- [Google - Site Reliability Engineering \(sre.google\)](#)
- [Cloud monitoring observability - Cloud Adoption Framework | Microsoft Learn](#)
- [Cloud monitoring service level objectives - Cloud Adoption Framework | Microsoft Learn](#)
- [Incident Response Guide: Best Practices | Squadcast](#)
- [New chat \(openai.com\)](#)
- [DevOps Terminology: Glossary of Common Terms - Sematext](#)
- [What is Observability? - YouTube](#)
- [Observability - Wikipedia](#)
- [Improving Observability and Testing In Production - YouTube](#)
- [Application logging and how to effectively use it | by Shubham Gupta | Ula Engineering | Medium](#)
- [Logging in C# - .NET | Microsoft Learn](#)
- [Inversion of Control Containers and the Dependency Injection pattern \(martinfowler.com\)](#)
- [Dependency injection - .NET | Microsoft Learn](#)