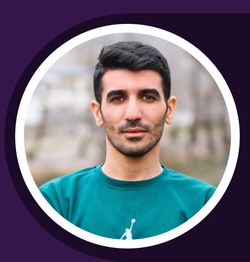
Logs

ASP.NET Core

Best practices

{#9.Tips}

By Example



Nabi Karampour
@thisisnabi

Logs

Is a record of events or messages that are generated by an application during its execution.

It’s for

- Understanding the behavior of applications
- Diagnosing issues
- Auditing events

#1 Use log frameworks

> Most programming languages have built-in logging capabilities but third-party libraries provide a robust, flexible, and scalable way to handle logging across different environments and applications.

Program.cs

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddSerilog((services, lc) => lc
       .ReadFrom.Configuration(builder.Configuration)
       .ReadFrom.Services(services)
       .Enrich.FromLogContext()
       .WriteTo.Console());
```

Tools like Serilog help you create structural logs and save you from reinventing the wheel.

#2 Use logging Levels

> It allows you to categorize and control the flow of log messages based on their severity and importance.

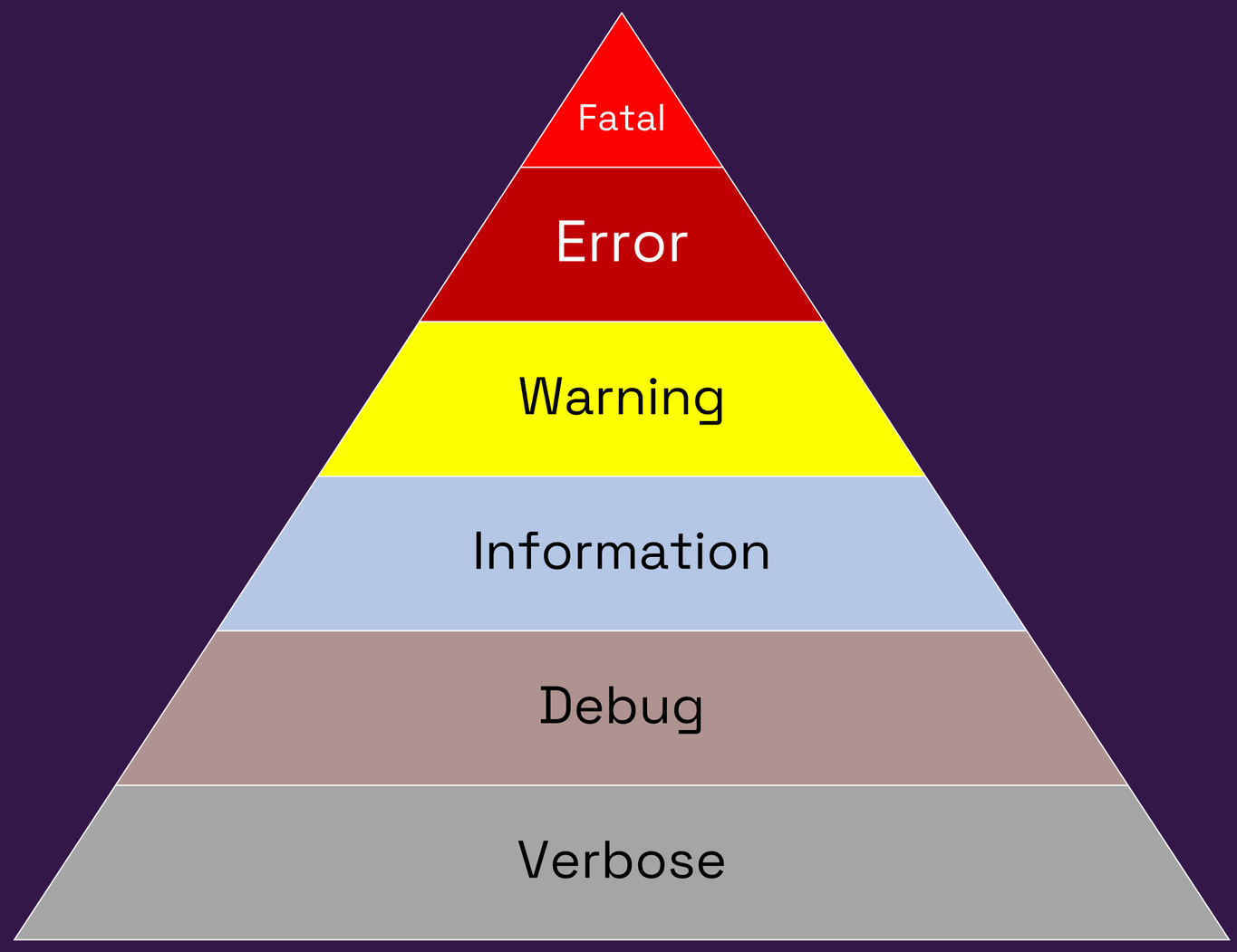
```
public IActionResult PerformOperation()
{
    _logger.LogDebug("Starting PerformOperation at {Time}", DateTime.UtcNow);

    try
    {
        if (new Random().Next(0, 2) == 0)
        {
            _logger.LogWarning("... as expected due to a non-critical issue.");
        }

        throw new InvalidOperationException("... occurred during the operation.");
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "An error occurred while performing the operation.");
    }
}
```

Serilog

Level



#3 Enable log levels

> for various environments, help manage the verbosity of logs according to the environment in which your application is running.

appsettings.Development.json

appsettings.Production.json

appsettings.Production.Json

```
{
  "Serilog": {
      "MinimumLevel": {
        "Default": "Debug",
        "Override": {
           "Shortener.Services": "Warning",
```

⚠ We will not see this log in the production environment

ShortenerServices.cs

```
namespace Shortener.Services;

public class ShortenUrlService
{

    public async Task<string> GenerateShortenUrlAsync(string longUrl)
    {
        var shortenCode = await GenerateCode(destinationUrl);

        _logger.LogInformation("Retriving request");
    ...
}
```

#4 Use Consistent formatting

> It ensures that logs are easy to read, parse, and analyze.

⛔ 👋

```
_logger.LogWarning("Failed to connect to DB!");
_logger.LogError("NullReferenceException occurred in the payment service");
_logger.LogDebug("Start of ProcessOrder method");
_logger.LogInformation("Order ID = 12345 was processed");
_logger.LogWarning("Low disk space detected - Drive C: has only 5% left.");
_logger.LogInformation("2024-08-09: User logged out");
```

Inconsistent formatting can make logs confusing and difficult to work.

✅ 😍

```
_logger.LogWarning("Failed to connect to database: DatabaseName.");
_logger.LogError("NullReferenceException [PS] during transaction processing.");
_logger.LogDebug("Starting method: ProcessOrder");
_logger.LogInformation("Order processed successfully. OrderId: 12345");
_logger.LogWarning("Low disk space detected. Drive: C, RemainingSpace: 5%");
_logger.LogInformation("User logout. Timestamp: 2024-08-09T14:23:45Z");

[PS] => PaymentService
```

#5 Use Structured logging

Involves capturing logs in a way that makes them easily queryable and analyzable.

```
[08:45:24 INF] User admin logged in successfully.
```

```
{
  "timestamp": "2024-08-02T12:00:00Z",
  "level": "INFO",
  "service": "user_service",
  "message": "User login successful",
  "userID": 12345
}
```

Structured

```
.WriteTo(s => s.Console(new ElasticsearchJsonFormatter()));
```

This allows logs to be easily parsed and analyzed by log management tools.

#6 Avoid string interpolation

> String interpolation creates a string by directly embedding variables into it before the log message is used. This means the full string with the variables filled in is made right away, so the logger only gets that complete string.

```
User user = GetUserFromAPI();
DateTime when = DateTime.UtcNow;

_logger.LogInformation($"Creating user: {user} at {when}");
```

⚠ New allocation even if this logging level is not enabled for output.

```
User user = GetUserFromAPI();
DateTime when = DateTime.UtcNow;

_logger.LogInformation("Creating user: {User} at {When}", user, when);
```

⚠ So if you use tools like ElasticSearch you can use ”User” as a search term.

#7 Mask sensitive data

> Logging sensitive information, such as passwords, credit card numbers, or personally identifiable information (PII), can lead to security risks.

⛔✋

```
_logger.LogInformation("User login attempt. Username: {Username}, Password: {Password}",
      username,
      Password);
```

Serilog.Enrichers.Sensitive

```
builder.Services.AddSerilog((services, lc) => lc
      .ReadFrom.Configuration(builder.Configuration)
      .ReadFrom.Services(services)
      .Enrich.FromLogContext()
      .Enrich.WithSensitiveDataMasking()
      .WriteTo.Console());
```

After

```
{
    "RenderedMessage": "User login attempt. Username: {Username}, Password: {***MASKED***}",
    "message": "User login attempt. Username: {Username}, Password: {Password}",
    "Properties.Password": "***MASKED***"
}
```

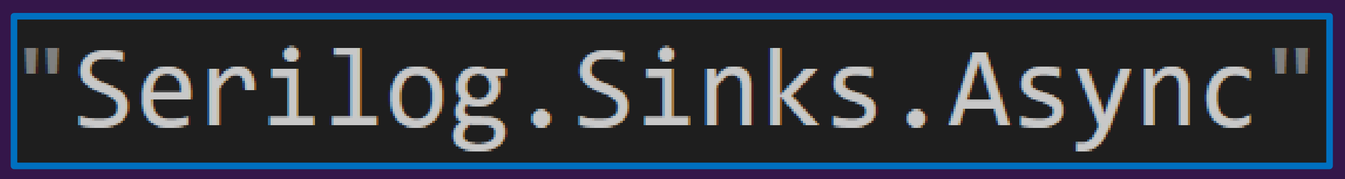
#8 Asynchronous Logging

> Which ensures that logging operations do not block or degrade the performance of the main application.

```
.WriteTo(s => s.Console(new ElasticsearchJsonFormatter()));
```

✅ 😍

```
configuration.ReadFrom.Configuration(context.Configuration)
             .Enrich.FromLogContext()
             .WriteTo.Async(s => s.Console(new ElasticsearchJsonFormatter()));
```



An asynchronous wrapper for Serilog sinks that logs on a background thread.

#9 Include Contextual Info

> Enhance log entries with additional context like request IDs, user IDs, session IDs. This helps in tracing and correlating related log entries.

```
[2024-08-02 12:00:00] INFO [user_service] [request_id=abc123] User login successful.
```

```
builder.Services.AddSerilog((services, lc) => lc
        .ReadFrom.Configuration(builder.Configuration)
        .Enrich.FromLogContext()
        .Enrich.WithProcessId()
        .Enrich.WithThreadId()
        .Enrich.WithExceptionDetails()
        .Enrich.WithCorrelationId()
        .Enrich.WithEnvironmentUserName();
```

Available enricher packages

The Serilog project provides:

- Serilog.Enrichers.Environment - `WithMachineName()` and `WithEnvironmentUserName()`
- Serilog.Enrichers.Process - `WithProcessId()`
- Serilog.Enrichers.Thread - `WithThreadId()`

Other interesting enrichers:

- Serilog.Web.Classic - `WithHttpRequestId()` and many other enrichers useful in classic ASP.NET applications
- Serilog.Exceptions - `WithExceptionDetails()` adds additional structured properties from exceptions
- Serilog.Enrichers.Demystify - `WithDemystifiedStackTraces()`
- Serilog.Enrichers.ClientInfo - `WithClientIp()`, `WithCorrelationId()` and `WithRequestHeader("header-name")` will add properties with client IP, correlation id and HTTP request header value
- Serilog.Enrichers.ExcelDna - `WithXllPath()` and many other enrichers useful in Excel-DNA add-ins
- Serilog.Enrichers.Sensitive - `WithSensitiveDataMasking()` masks sensitive data in log events
- Serilog.Enrichers.GlobalLogContext - `FromGlobalLogContext()` adds properties from the "global context" that can be added dynamically

Or even you can make custom enricher.

```
namespace ResidentAPI.CustomEnrichers
{
    public class TraceIdContextEnricher : ILogEventEnricher
```

Repost, so your friends can learn too.

Let’s follow