The Future of Structured Logging with NLog in .NET Ecosystems

AzraJabeen Mohamed Ali

Azra.jbn@gmail.com
Independent researcher, California,USA

Abstract
This research explores the evolution of structured logging with NLog, examining its current capabilities, integration with .NET Core and .NET 5/6 ecosystems, and its role in improving application monitoring, debugging, and troubleshooting.The study investigates the core features of NLog that facilitate structured logging, including the use of JSON-based layouts, logging targets, and advanced filtering techniques. It also delves into the integration of NLog with cloud platforms, microservices, and distributed systems, highlighting the challenges and opportunities of implementing structured logging in these environments. Additionally, the research assesses how NLog behaves in terms of flexibility, performance, and scalability.By identifying emerging trends and potential improvements in structured logging, the paper outlines best practices for leveraging NLog to meet the needs of modern software architectures, ensuring reliable error tracking, and enhancing observability. The findings aim to provide insights for .NET developers and architects on how to adopt and maximize the potential of structured logging with NLog in the future of software development.

Keywords: NLog, logging,.Net, ecosystem, scalability, debugging, error tracking

1. Introduction

In the evolving landscape of modern software development, logging has become an indispensable tool for ensuring application reliability, security, and performance. As systems grow more complex and distributed, the need for structured logging has gained significant attention. Structured logging offers a more systematic approach to capturing log data, organizing it in a machine-readable format that enhances the traceability, analysis, and monitoring of applications. This approach is particularly crucial in dynamic environments such as cloud-native systems, microservices architectures, and enterprise-level applications.

The .NET ecosystem, which encompasses a wide range of applications from desktop software to large-scale web services, has seen a shift towards more efficient and scalable logging solutions. Among these, NLog stands out as a highly popular logging framework, known for its flexibility, extensibility, and robust integration capabilities. NLog's support for structured logging has made it an attractive choice for developers looking to implement detailed, consistent, and easily searchable logs across their applications.

As organizations increasingly migrate to microservices architectures, embrace DevOps practices, and adopt cloud-first strategies, the need for effective logging solutions is more pressing than ever. Structured logging with NLog offers a pathway to better observability, improved debugging, and more insightful operational analytics. However, despite its growing adoption, challenges remain in fully optimizing structured logging within the .NET ecosystem. These include complexities in configuration, integration with cloud platforms, handling high volumes of log data, and ensuring compatibility with other tools and services.

NLog:

NLog is a popular and flexible logging framework for .NET applications, used to log and monitor information about system behavior, errors, and application flow. It provides an easy-to-use mechanism for logging events with the ability to route logs to various outputs (or targets), including text files, databases, cloud services, and more.

Structured logging refers to the practice of capturing logs in a format that can be easily parsed and analyzed, typically using a machine-readable structure such as JSON or XML. This contrasts with traditional logging, which often uses plain text formats that are harder to parse programmatically. Structured logging enhances the ability to query, filter, and analyze logs, making it particularly valuable in complex applications, microservices, and cloud environments.

NLog, as a flexible and powerful logging framework for .NET, offers full support for structured logging. This allows developers to log information in a structured format that can be easily consumed by log aggregation and analysis tools, such as Elasticsearch, Splunk, or cloud-native monitoring services.

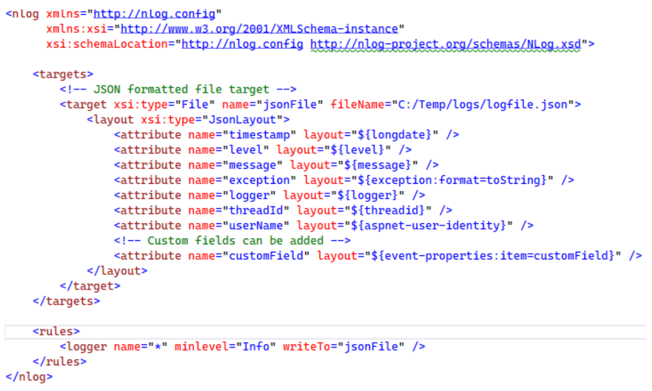
How to Implement Structured Logging with NLog:

After downloading and opening Visual Studio, right-click the project and select Manage Nuget Packages. Search NLog under browse and choose the NLog and click Install.

NLog Configuration:

The configuration can be done using an XML file, programmatically, or via JSON. Below is an example of how to configure NLog with structured logging in a JSON format using an XML configuration file. In this configuration, logs are written in JSON format to a file. The layout specifies various attributes like timestamp, log level, message, exception, and custom fields Fig-1.

Fig-1:



```
<nlog xmlns="http://nlog.config"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://nlog.config http://nlog-project.org/schemas/NLog.xsd">

    <targets>
        <!-- JSON formatted file target -->
        <target xsi:type="File" name="jsonFile" fileName="C:/Temp/logs/logfile.json">
            <layout xsi:type="JsonLayout">
                <attribute name="timestamp" layout="${longdate}" />
                <attribute name="level" layout="${level}" />
                <attribute name="message" layout="${message}" />
                <attribute name="exception" layout="${exception:format=toString}" />
                <attribute name="logger" layout="${logger}" />
                <attribute name="threadId" layout="${threadid}" />
                <attribute name="userName" layout="${aspnet-user-identity}" />
                <!-- Custom fields can be added -->
                <attribute name="customField" layout="${event-properties:item=customField}" />
            </layout>
        </target>
    </targets>

    <rules>
        <logger name="*" minlevel="Info" writeTo="jsonFile" />
    </rules>
</nlog>
```

The logs will be written to a file named logfile.json located in the logs folder. The log entries will be formatted in JSON with the following attributes:

- **timestamp:** Logs the current date and time using the ${longdate} layout renderer.
- **level:** Captures the log level (e.g., Info, Error, etc.).
- **message:** Logs the actual log message.
- **exception:** If an exception is logged, this field will capture the exception's string representation using the ${exception:format=toString} layout renderer.
- **logger:** The name of the logger that generated the log entry.
- **threadId:** Captures the thread ID for multithreaded applications.

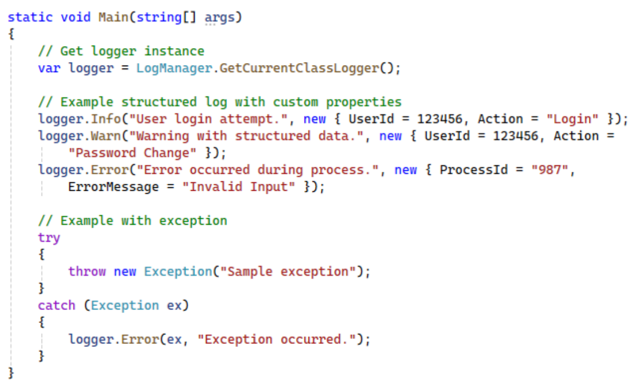
- **userName:** Logs the user identity, typically used in web applications (ASP.NET).
- **customField:** A custom field added to logs, which can be set dynamically during logging.

Loggingrules ensure that all loggers, starting from the Info log level, will output to the jsonFile target. It is possible to modify the minlevel to control which log levels are captured. For example, use Error for only errors or Debug for more detailed logs.

Structured logging code in C#:

This code Fig-2 demonstrates how to use NLog for structured logging, logging custom properties, and handling exceptions. By using a JSON format, these logs can easily be consumed by log management systems (e.g., ELK stack, Splunk), providing a more organized and machine-readable logging solution.The provided C# code demonstrates logging with NLog, including structured logging with custom properties and logging of exceptions.Custom properties are passed to the logger using anonymous types.This logs the message "User login attempt." with additional structured properties like UserId and Action.Below code captures the exception details and logs them alongside the message "Exception occurred.". The log levels (Info, Warn, Error) are used to indicate the severity of each log entry.

Fig-2:

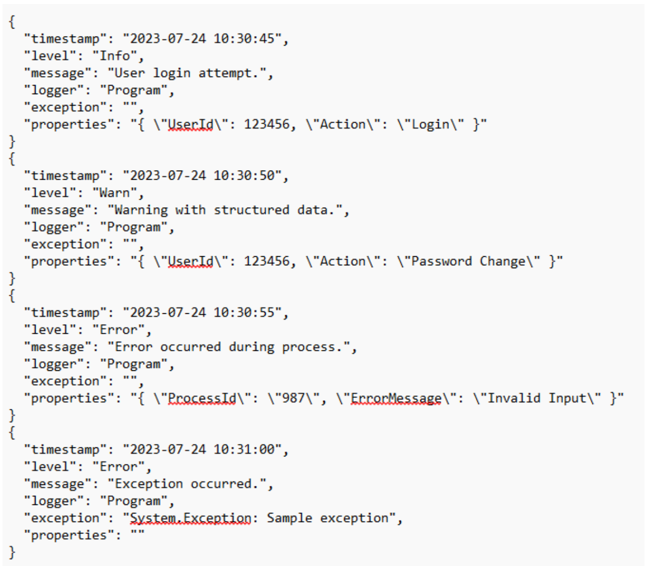


```
static void Main(string[] args)
{
    // Get logger instance
    var logger = LogManager.GetCurrentClassLogger();

    // Example structured log with custom properties
    logger.Info("User login attempt.", new { UserId = 123456, Action = "Login" });
    logger.Warn("Warning with structured data.", new { UserId = 123456, Action =
        "Password Change" });
    logger.Error("Error occurred during process.", new { ProcessId = "987",
        ErrorMessage = "Invalid Input" });

    // Example with exception
    try
    {
        throw new Exception("Sample exception");
    }
    catch (Exception ex)
    {
        logger.Error(ex, "Exception occurred.");
    }
}
```

Output:

The logs in logfile.json look like Fif-3:

Fig-3



```
{
  "timestamp": "2023-07-24 10:30:45",
  "level": "Info",
  "message": "User login attempt.",
  "logger": "Program",
  "exception": "",
  "properties": "{ \"UserId\": 123456, \"Action\": \"Login\" }"
}
{
  "timestamp": "2023-07-24 10:30:50",
  "level": "Warn",
  "message": "Warning with structured data.",
  "logger": "Program",
  "exception": "",
  "properties": "{ \"UserId\": 123456, \"Action\": \"Password Change\" }"
}
{
  "timestamp": "2023-07-24 10:30:55",
  "level": "Error",
  "message": "Error occurred during process.",
  "logger": "Program",
  "exception": "",
  "properties": "{ \"ProcessId\": \"987\", \"ErrorMessage\": \"Invalid Input\" }"
}
{
  "timestamp": "2023-07-24 10:31:00",
  "level": "Error",
  "message": "Exception occurred.",
  "logger": "Program",
  "exception": "System.Exception: Sample exception",
  "properties": ""
}
```

Default Log level systems:

NLog provides a robust log level system that helps categorize the severity of log messages. The log levels represent the importance or severity of an event or message being logged. NLog's default log levels help developers determine the appropriate level of detail to log based on the context of the event, enabling better log management and analysis. Below are the default log levels in NLog, listed from the most verbose (least severe) to the most critical (most severe):

1. **Trace:**
   This is the most detailed and verbose log level. It is typically used for low-level debugging and diagnostic information.It is used for tracing the flow of an application and recording every step of the process, such as method calls, variable values, or detailed interactions with system components. Log Level number is 0 and it is typically used In-depth troubleshooting during development or when it is needed to track detailed application behavior.

2. **Debug:**
   Debug messages are used for debugging purposes during development or in non-production environments. It contains useful information for developers to trace issues. It is used for logging detailed information that helps debug issues but is not necessarily critical. Log level number is 1. It is typically used during development and debugging sessions to understand the application's internal behavior.

3. **Info:**
   Informational messages provide details about the general flow of the application. These are typically less verbose than debug or trace logs but still contain helpful information. It is used to log general application flow, successful events, or actions that do not indicate any issues. Often used in production to track regular operations. Log level number is 2. It is typically used for logging normal application operations such as service starts, successful transactions, or user actions.

4. **Warn:**
   Warning messages indicate potential issues or unusual situations that are not necessarily errors but could lead to problems if not addressed. Warnings typically do not interrupt the application's flow.

Benefits of Structured Logging with NLog:

The key benefits are:

1. **Improved Log Analysis and Searchability:**
   - **Machine-Readable Format:** Structured logging ensures that logs are stored in machine-readable formats like JSON or XML, as opposed to unstructured text logs. This makes logs easily searchable and analyzable using tools like Elasticsearch, Splunk, or other log management systems.
   - **Efficient Querying**: With structured logs, you can query specific fields such as log level, timestamp, user ID, or custom application context, which would be difficult to achieve with plain-text logs.
   - **Enhanced Filtering:** When logs are structured, you can filter by various attributes (e.g., log level, exception type, user session), which simplifies the process of identifying patterns and diagnosing issues.

2. **Enhanced Debugging and Troubleshooting:**
    - **Rich Contextual Information:** Structured logging allows developers to log additional contextual data, such as user IDs, transaction IDs, IP addresses, or custom application-specific properties. This contextual information significantly improves troubleshooting by allowing engineers to track down the root cause of issues faster.
    - **Correlation Across Services:** In microservice-based architectures, structured logging allows you to correlate logs from different services using unique identifiers, such as trace IDs or request IDs, which simplifies identifying issues that span multiple services.
    - **Clearer Error Tracking:** Structured logging allows exceptions to be logged in a way that makes it easy to analyze stack traces, error messages, and relevant context in a consistent format, enhancing error diagnosis.

3. **Better Monitoring and Observability:**
    - **Real-Time Insights:** Structured logs provide the necessary context (such as request IDs, user sessions, or error types) to effectively monitor application behavior in real time. This leads to better visibility into how the application is functioning, especially in a production environment.
    - **Integration with Observability Tools:** Structured logs can be easily integrated with monitoring tools like Prometheus, Grafana, or cloud-based monitoring services. This facilitates automated alerting, dashboards, and visualizations that improve operational awareness.
    - **Scalability:** As applications scale, especially in cloud or microservices environments, structured logging allows for efficient log aggregation and analysis. You can quickly analyze logs across multiple services, helping to detect and resolve issues that affect the overall system performance.

4. **Easier Compliance and Auditability:**
    - **Audit Trails:** Structured logs can store detailed information about user actions, system events, and changes within the application. This makes it easier to maintain an audit trail, which is crucial for regulatory compliance in industries such as finance, healthcare, and e-commerce.
    - **Standardized Logging:** By using structured logging, you ensure that logs across the entire application are consistent, making it easier to review and analyze logs for compliance or auditing purposes. This standardization reduces the chances of missing critical events or information.

5. **Greater Flexibility and Customization:**
    - **Custom Fields:** NLog allows developers to include custom fields in logs, providing flexibility in capturing application-specific data. Whether it's user information, transaction IDs, or any custom metadata, you can define exactly what data should be logged, making the logs more meaningful and contextually rich.
    - **Custom Layouts:** NLog offers a variety of layout renderers (such as ${date}, ${level}, ${message}), and custom layout options, allowing developers to control the format and structure of the log data. This ensures that logs are tailored to specific needs and output formats (e.g., JSON for structured logging).

- **Multiple Targets:** NLog supports outputting logs to multiple targets simultaneously. With structured logging, you can send logs in a structured format to various destinations, such as databases, log management services, or files, without losing the structure or context of the logs.

6. **Reduced Log Volume and Improved Performance:**
   - **Efficient Logging:** With structured logging, logs can be more concise and efficient. By only capturing necessary details (via custom fields and layout settings), you reduce the volume of extraneous log data, which is particularly useful in high-traffic applications where log volume can quickly become overwhelming.
   - **Asynchronous Logging:** NLog supports asynchronous logging, meaning logs can be written in the background without blocking the main application. This ensures that logging overhead does not impact the performance of the application, even when structured data is being written.

7. **Seamless Integration with DevOps and CI/CD Pipelines:**
   - **Automation and Continuous Monitoring:** With structured logs, integration with DevOps pipelines becomes smoother. Logs can be sent to centralized logging systems that are continuously monitored, enabling rapid detection of issues in development, staging, or production environments.
   - **Log Aggregation:** Structured logging facilitates the aggregation of logs from different stages of the development and deployment pipeline, helping teams quickly identify issues early in the CI/CD pipeline and avoid problems in production.

8. **Long-Term Data Retention and Analysis:**
   - **Efficient Data Storage:** Structured logs are easier to store, index, and retrieve over the long term. As logs are generated in a standardized format (e.g., JSON), they can be indexed for faster searches and can be retained for historical analysis. This is particularly important for post-mortem analysis or future audits.
   - **Data Mining and Trend Analysis:** With structured logs, it’s easier to perform trend analysis over time. For instance, you can look for patterns in user behavior, error rates, or system performance, providing valuable insights for optimizing application performance or identifying areas for improvement.

Conclusion:

The landscape of modern software development is rapidly evolving, and with it, the need for effective and efficient logging practices. As applications grow in complexity, especially with the rise of microservices, cloud-native systems, and distributed architectures, traditional logging methods no longer suffice. Structured logging has emerged as a critical solution, offering enhanced traceability, searchability, and actionable insights for both development and operations teams.

NLog, with its robust support for structured logging, is well-positioned to meet the needs of the .NET ecosystem. By providing a flexible and powerful framework that allows for the capture of logs in machine-readable formats like JSON, NLog empowers developers to gather rich, context-aware logs that can be easily integrated with modern monitoring, alerting, and log aggregation systems. Its extensibility, high performance, and ability to seamlessly handle structured data make it an ideal choice for building observability into .NET applications.

As the demand for observability increases, NLog’s structured logging capabilities will continue to evolve. Future developments are likely to focus on deeper integration with emerging cloud platforms, real-time analytics, and the enhancement of distributed tracing for microservices. Additionally, NLog's ability to handle high volumes of data with minimal impact on application performance will be crucial as applications scale.

In conclusion, the future of structured logging with NLog in .NET ecosystems is promising. It will play an essential role in enhancing the maintainability, reliability, and performance of modern applications. By adopting structured logging practices and leveraging NLog's capabilities, developers can gain deeper visibility into their systems, improve troubleshooting, and streamline the management of large-scale, distributed applications. As technology continues to advance, the role of structured logging in ensuring effective application monitoring and operational efficiency will only grow more vital.

References

[1] Harpreet Singh, “Basic Understanding Of NLog” https://www.c-sharpcorner.com/article/basic-understanding-of-nlog/(May 15, 2023)

[2] Thomas Ardal, “NLog Tutorial - The essential guide for logging from C#” https://blog.elmah.io/nlog-tutorial-the-essential-guide-for-logging-from-csharp/(Mar31, 2020)

[3] Grant Winney “How to log messages to multiple targets with NLog“ https://grantwinney.com/how-to-log-messages-to-multiple-targets-with-nlog/ (Jul 1, 2023)

[4] Grant Winney “How to log errors in WinForms using NLog“ https://grantwinney.com/log-errors-in-winforms-with-nlog/(Oct 9, 2021)

[5] Seq “Using NLog” https://docs.datalust.co/v5.1/docs/using-nlog(2019)

[6] C# Forums “NLog logging“ https://csharpforums.net/threads/nlog-logging.4777/ (Feb 22, 2019)

[7] Jaroslaw Kowalski “Introduction to NLog” https://www.codeproject.com/Articles/10631/Introduction-to-NLog(Jul 13, 2006)

[8] Jeremy Lindsay“How to use NLog or Serilog with C# in ASP.NET Core”https://jeremylindsayni.wordpress.com/2016/03/27/how-to-use-nlog-or-serilog-with-c-in-asp-net-core/ (Mar 27, 2016)

[9] Kanti Kalyan Arumilli“Programatically configuring NLog in C#” https://www.alightservices.com/2022/07/11/programatically-configuring-nlog-in-c/ (Jul 11, 2022)

[10] Kirk Patrick junsay“C#: Implementing NLog in your application” https://codesandchips.com/2016/07/15/c-implementing-nlog-in-your-application/(Jul 15, 2016)

[11] Ivan Yakimov “NLog: Rules and filters” https://www.codeproject.com/Articles/4051307/NLog-Rules-and-filters (May 6, 2019)

[12] Github.com “How to use structured logging“ https://github.com/NLog/NLog/wiki/How-to-use-structured-logging (Jan 1, 2023)