**Especificación Técnica**

**Solutions.Utility.AppLogger**

**Versión: 1.0.0**

**Fecha: 22/07/2025**

Contenido

[Solutions.Utility.AppLogger 3](#_Toc204103506)

[1. Introducción 3](#_Toc204103507)

[2. Arquitectura del Proyecto 3](#_Toc204103508)

[2.1. Interfaz ILogger 3](#_Toc204103509)

[2.2. Clase Logger 4](#_Toc204103510)

[3. Configuración y Uso 4](#_Toc204103511)

[3.1. Configuración Inicial 4](#_Toc204103512)

[3.2. Uso en la Aplicación 5](#_Toc204103513)

[4. Consideraciones Adicionales 6](#_Toc204103514)

[5. Diagrama de Clases (Conceptual) 6](#_Toc204103515)

[6. Próximos Pasos 8](#_Toc204103516)

# Solutions.Utility.AppLogger

## 1. Introducción

El proyecto Solutions.Utility.AppLogger es una biblioteca C# diseñada para facilitar la **generación y gestión de archivos de log** en aplicaciones. Su objetivo principal es proporcionar una **interfaz estandarizada y fácil de usar** para registrar eventos de diferentes niveles de severidad (Info, Debug, Warn, Error, Fatal), permitiendo así el seguimiento y la depuración de los flujos de ejecución de los programas.

## 2. Arquitectura del Proyecto

El proyecto está compuesto por los siguientes elementos principales:

### 2.1. Interfaz ILogger

La interfaz ILogger define el contrato para cualquier clase que implemente la funcionalidad de logging. Esto permite una **arquitectura desacoplada**, donde la implementación concreta del logger puede ser cambiada sin afectar el código que lo utiliza.

**Métodos de Configuración:**

* public void AddFile(string file): Especifica el **nombre del archivo de log**.
* public void AddPath(string path): Define la **ruta del directorio** donde se guardarán los archivos de log.
* public void DateFormatLog(string format): Establece el **formato de fecha y hora** que se utilizará dentro de las entradas del log.

**Métodos de Registro de Mensajes (sin formato):**

* void Info(string message): Registra mensajes informativos sobre el flujo normal de la aplicación.
* void Debug(string message): Registra mensajes detallados para propósitos de depuración.
* void Warn(string message): Registra advertencias sobre posibles problemas o situaciones no críticas.
* void Error(string message): Registra errores que indican fallos en partes de la aplicación.
* void Fatal(string message): Registra errores críticos que pueden llevar al cierre de la aplicación.

**Métodos de Registro de Mensajes (con formato):**

* void InfoFormat(string message, params object[] args): Permite formatear mensajes informativos utilizando argumentos.
* void DebugFormat(string message, params object[] args): Permite formatear mensajes de depuración.
* void WarnFormat(string message, params object[] args): Permite formatear mensajes de advertencia.
* void ErrorFormat(string message, params object[] args): Permite formatear mensajes de error.
* void FatalFormat(string message, params object[] args): Permite formatear mensajes fatales.

### 2.2. Clase Logger

La clase Logger es la **implementación concreta de la interfaz ILogger**. Contendrá la lógica para escribir los mensajes en el archivo de log especificado, gestionando la ruta, el nombre del archivo y el formato de fecha.

## 3. Configuración y Uso

La biblioteca AppLogger está diseñada para ser **fácilmente integrable** en aplicaciones C#, especialmente aquellas que utilizan inyección de dependencias (como las aplicaciones ASP.NET Core).

### 3.1. Configuración Inicial

La configuración inicial del logger se realiza a través de un método de extensión para IServiceCollection, lo que facilita su adición al **contenedor de inyección de dependencias**.

C#

public static IServiceCollection AddLogger(this IServiceCollection services)

{

// Se asume la existencia de una clase LoggerAdapter que encapsula la implementación de Logger.

// Esto sugiere una posible capa de abstracción adicional o una adaptación a un framework de logging existente.

IAppLogger logging = new LoggerAdapter();

// Configuración de la ruta del directorio de logs

logging.AddPath($"C:\\Logs");

// Configuración del nombre del archivo de log, incluyendo la fecha actual

logging.AddFile($"LOG\_Main.Service.Api\_" + DateTime.Now.ToString("yyyyMMdd") + ".log");

// Se registra la instancia de IAppLogger como un Singleton en el contenedor de servicios.

// Esto significa que la misma instancia del logger será reutilizada en toda la aplicación.

services.AddSingleton(logging);

return services;

}

**Puntos Clave de la Configuración:**

* **LoggerAdapter**: La presencia de LoggerAdapter sugiere que la implementación Logger podría estar siendo adaptada o encapsulada, quizás para integrar con un sistema de logging más complejo o para proporcionar una capa adicional de abstracción. Se deberá especificar cómo LoggerAdapter se relaciona con la clase Logger (e.g., si LoggerAdapter contiene una instancia de Logger o si LoggerAdapter directamente implementa ILogger y actúa como la clase Logger).
* **Ruta y Nombre del Archivo**: La configuración permite establecer dinámicamente la ruta y el nombre del archivo de log, incluyendo la fecha actual en el nombre del archivo para facilitar la organización diaria de los logs.
* **AddSingleton**: Registrar el logger como Singleton asegura que solo una instancia del logger se cree durante el ciclo de vida de la aplicación, lo cual es eficiente para la gestión de recursos al escribir en archivos.

### 3.2. Uso en la Aplicación

Una vez configurado, el ILogger puede ser **inyectado en cualquier clase** donde se necesite registrar eventos.

C#

// Ejemplo de inyección en un constructor

public class MyService

{

private readonly ILogger \_logger;

public MyService(ILogger logger)

{

\_logger = logger;

}

public void DoSomething()

{

\_logger.Info("Iniciando operación DoSomething.");

try

{

// Lógica de negocio

\_logger.DebugFormat("Procesando item: {0}", itemId);

}

catch (Exception ex)

{

\_logger.ErrorFormat(ex, "Ocurrió un error en DoSomething: {0}", ex.Message); // Asumiendo que ErrorFormat puede manejar excepciones

\_logger.Fatal("Error fatal, la aplicación podría no recuperarse.");

}

\_logger.Info("Operación DoSomething finalizada.");

}

}

## 4. Consideraciones Adicionales

* **Manejo de Errores en el Logging**: La implementación de la clase Logger debe incluir un robusto manejo de errores para situaciones como permisos de escritura insuficientes en la ruta del log, disco lleno, o nombres de archivo inválidos.
* **Formato de Salida del Log**: Se debe definir claramente el formato de cada entrada de log (por ejemplo, [FECHA HORA] [NIVEL] - MENSAJE).
* **Rotación de Logs**: Para evitar que los archivos de log crezcan indefinidamente, se podría considerar implementar una **política de rotación de logs** (por tamaño, por tiempo, etc.).
* **Logging Asíncrono**: Para aplicaciones de alto rendimiento, el logging asíncrono podría mejorar el rendimiento general de la aplicación al no bloquear el hilo principal durante las operaciones de escritura en disco.
* **Configuración a través de Archivos**: En lugar de codificar las rutas y nombres de archivo directamente, se podría permitir la **configuración a través de archivos** (ej., appsettings.json en .NET Core) para una mayor flexibilidad.
* **Nivel Mínimo de Log**: Podría ser útil permitir la configuración de un **nivel mínimo de log** (ej., solo registrar errores y fatales en producción) para controlar la verbosidad de los logs.

## 5. Diagrama de Clases (Conceptual)

Fragmento de código

classDiagram

direction LR

interface ILogger {

+void AddFile(string file)

+void AddPath(string path)

+void DateFormatLog(string format)

+void Info(string message)

+void Debug(string message)

+void Warn(string message)

+void Error(string message)

+void Fatal(string message)

+void InfoFormat(string message, params object[] args)

+void DebugFormat(string message, params object[] args)

+void WarnFormat(string message, params object[] args)

+void ErrorFormat(string message, params object[] args)

+void FatalFormat(string message, params object[] args)

}

class Logger {

-string \_filePath

-string \_fileName

-string \_dateFormat

+Logger()

+AddFile(string file)

+AddPath(string path)

+DateFormatLog(string format)

+Info(string message)

+Debug(string message)

+Warn(string message)

+Error(string message)

+Fatal(string message)

+InfoFormat(string message, params object[] args)

+DebugFormat(string message, params object[] args)

+WarnFormat(string message, params object[] args)

+ErrorFormat(string message, params object[] args)

+FatalFormat(string message, params object[] args)

-WriteLog(string level, string message)

}

class LoggerAdapter {

-ILogger \_innerLogger

+LoggerAdapter()

+AddFile(string file)

+AddPath(string path)

+DateFormatLog(string format)

+Info(string message)

+Debug(string message)

+Warn(string message)

+Error(string message)

+Fatal(string message)

+InfoFormat(string message, params object[] args)

+DebugFormat(string message, params object[] args)

+WarnFormat(string message, params object[] args)

+ErrorFormat(string message, params object[] args)

+FatalFormat(string message, params object[] args)

}

ILogger <|.. Logger : implements

ILogger <|.. LoggerAdapter : implements

LoggerAdapter --> Logger : aggregates (optional, if Adapter wraps Logger)

## 6. Próximos Pasos

1. **Implementación de la clase Logger**: Desarrollar la lógica interna de escritura de archivos.
2. **Implementación de la clase LoggerAdapter**: Definir cómo se relaciona con Logger y si agrega funcionalidad adicional o solo actúa como un passthrough.
3. **Pruebas Unitarias**: Crear pruebas para asegurar la correcta funcionalidad de cada método de logging y configuración.
4. **Ejemplos de Uso**: Proporcionar ejemplos claros de cómo configurar y utilizar el logger en diferentes escenarios.