**Especificación Funcional**

**Solutions.Utility.AppLogger**

**Versión: 1.0.0**

**Fecha: 22/07/2025**

Contenido

[1. Descripción General 3](#_Toc204091916)

[2. Objetivos 3](#_Toc204091917)

[3. Alcance 3](#_Toc204091918)

[4. Funcionalidades 3](#_Toc204091919)

[5. Configuración de Ejemplo 3](#_Toc204091920)

[6. Requisitos Funcionales 4](#_Toc204091921)

[7. Requisitos No Funcionales 4](#_Toc204091922)

[8. Restricciones 4](#_Toc204091923)

[9. Especificación Funcional 4](#_Toc204091924)

[1. Interfaz ILogger.cs 4](#_Toc204091925)

[2. Clase Logger.cs 6](#_Toc204091926)

[3. Clase LoggerAdapter.cs 9](#_Toc204091927)

[4. Clase de Extensión para IServiceCollection (por ejemplo, LoggerServiceExtensions.cs) 10](#_Toc204091928)

[5. Ejemplo de Uso en una Aplicación (por ejemplo, Program.cs para una Consola o Startup.cs para ASP.NET Core) 11](#_Toc204091929)

[Cómo Usar Este Documento Funcional: 12](#_Toc204091930)

Especificación Funcional

# 1. Descripción General

El proyecto Solutions.Utility.AppLogger es una librería destinada a facilitar la generación de logs en aplicaciones desarrolladas en C#. Provee una interfaz común (ILogger) para registrar información de diagnóstico, advertencias, errores y otros eventos relevantes en archivos de texto, permitiendo una trazabilidad eficaz del comportamiento de los sistemas que lo utilizan.

# 2. Objetivos

- Proveer una utilidad reutilizable para generación de logs.  
- Permitir la configuración dinámica de nombre de archivo, ruta y formato de fecha.  
- Facilitar la integración mediante inyección de dependencias.

# 3. Alcance

La utilidad puede ser utilizada en cualquier proyecto basado en .NET que requiera generar archivos de log. Permite la configuración de ruta de almacenamiento, nombre de archivo, y ofrece múltiples niveles de log como Info, Debug, Warn, Error y Fatal.

# 4. Funcionalidades

- Registrar mensajes con diferentes niveles de importancia.

- Formatear mensajes con parámetros dinámicos.

- Configurar la ruta y nombre del archivo log.

- Personalizar el formato de fecha.

- Integración con inyección de dependencias.

# 5. Configuración de Ejemplo

public static IServiceCollection AddLogger(this IServiceCollection services)  
{  
 IAppLogger logging = new LoggerAdapter();  
 logging.AddPath("C:\\Logs");  
 logging.AddFile("LOG\_Main.Service.Api\_" + DateTime.Now.ToString("yyyyMMdd") + ".log");  
 services.AddSingleton(logging);  
 return services;  
}

# 6. Requisitos Funcionales

RF01: Permitir la configuración de la ruta del archivo log.

RF02: Permitir la configuración del nombre del archivo log.

RF03: Registrar mensajes con distintos niveles de importancia.

RF04: Permitir formateo de mensajes con argumentos variables.

RF05: Exponer los métodos a través de una interfaz común (ILogger).

# 7. Requisitos No Funcionales

RNF01: El componente debe ser reutilizable y desacoplado.

RNF02: Los archivos de log deben ser legibles por humanos.

RNF03: La utilidad debe ser compatible con múltiples entornos .NET.

# 8. Restricciones

- No incluye almacenamiento en base de datos ni nube.

- Requiere permisos de escritura en el sistema de archivos.

# 9. Especificación Funcional

## 1. Interfaz ILogger.cs

Este archivo define el contrato para el sistema de logging.

using System;

namespace Solutions.Utility.AppLogger

{

public interface ILogger

{

/// <summary>

/// Agrega el nombre del archivo de log.

/// </summary>

/// <param name="file">El nombre del archivo (ej. "MiApp.log").</param>

void AddFile(string file);

/// <summary>

/// Agrega la ruta del directorio donde se guardarán los archivos de log.

/// </summary>

/// <param name="path">La ruta del directorio (ej. "C:\\Logs").</param>

void AddPath(string path);

/// <summary>

/// Establece el formato de fecha y hora para las entradas del log.

/// </summary>

/// <param name="format">El formato de fecha y hora (ej. "yyyy-MM-dd HH:mm:ss").</param>

void DateFormatLog(string format);

/// <summary>

/// Registra un mensaje informativo.

/// </summary>

/// <param name="message">El mensaje a registrar.</param>

void Info(string message);

/// <summary>

/// Registra un mensaje de depuración.

/// </summary>

/// <param name="message">El mensaje a registrar.</param>

void Debug(string message);

/// <summary>

/// Registra un mensaje de advertencia.

/// </summary>

/// <param name="message">El mensaje a registrar.</param>

void Warn(string message);

/// <summary>

/// Registra un mensaje de error.

/// </summary>

/// <param name="message">El mensaje a registrar.</param>

void Error(string message);

/// <summary>

/// Registra un mensaje fatal.

/// </summary>

/// <param name="message">El mensaje a registrar.</param>

void Fatal(string message);

/// <summary>

/// Registra un mensaje informativo con formato.

/// </summary>

/// <param name="message">El mensaje de formato.</param>

/// <param name="args">Argumentos para el formato del mensaje.</param>

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

/// <summary>

/// Registra un mensaje de depuración con formato.

/// </summary>

/// <param name="message">El mensaje de formato.</param>

/// <param name="args">Argumentos para el formato del mensaje.</param>

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

/// <summary>

/// Registra un mensaje de advertencia con formato.

/// </summary>

/// <param name="message">El mensaje de formato.</param>

/// <param name="args">Argumentos para el formato del mensaje.</param>

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

/// <summary>

/// Registra un mensaje de error con formato.

/// </summary>

/// <param name="message">El mensaje de formato.</param>

/// <param name="args">Argumentos para el formato del mensaje.</param>

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

/// <summary>

/// Registra un mensaje fatal con formato.

/// </summary>

/// <param name="message">El mensaje de formato.</param>

/// <param name="args">Argumentos para el formato del mensaje.</param>

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

}

}

## 2. Clase Logger.cs

Esta clase implementa la interfaz ILogger y se encarga de escribir los mensajes en un archivo.

using System;

using System.IO;

using System.Text;

namespace Solutions.Utility.AppLogger

{

public class Logger : ILogger

{

private string \_logFilePath;

private string \_logFileName;

private string \_dateFormat = "yyyy-MM-dd HH:mm:ss.fff"; // Formato por defecto

// Propiedades para almacenar la ruta y el nombre base del archivo

private string \_basePath;

private string \_baseFile;

public Logger()

{

// Constructor, si se necesita alguna inicialización por defecto

}

public void AddFile(string file)

{

\_baseFile = file; // Guarda el nombre base del archivo

UpdateLogFilePath();

}

public void AddPath(string path)

{

\_basePath = path; // Guarda la ruta base

UpdateLogFilePath();

}

public void DateFormatLog(string format)

{

\_dateFormat = format;

}

/// <summary>

/// Actualiza la ruta completa del archivo de log combinando la ruta y el nombre del archivo.

/// </summary>

private void UpdateLogFilePath()

{

if (!string.IsNullOrEmpty(\_basePath) && !string.IsNullOrEmpty(\_baseFile))

{

// Asegura que el directorio exista

try

{

if (!Directory.Exists(\_basePath))

{

Directory.CreateDirectory(\_basePath);

}

\_logFilePath = Path.Combine(\_basePath, \_baseFile);

}

catch (Exception ex)

{

// En un escenario real, aquí se podría loguear este error a la consola o a un visor de eventos

Console.WriteLine($"Error al crear el directorio de logs o combinar la ruta: {ex.Message}");

\_logFilePath = null; // Invalida la ruta si hay un error

}

}

}

private void WriteLog(string level, string message)

{

if (string.IsNullOrEmpty(\_logFilePath))

{

Console.WriteLine($"Error: Ruta del archivo de log no configurada o inválida para el mensaje: {message}");

return;

}

try

{

string logEntry = $"{DateTime.Now.ToString(\_dateFormat)} [{level}] - {message}";

// Usamos File.AppendAllText para añadir al final del archivo.

// Esto crea el archivo si no existe.

File.AppendAllText(\_logFilePath, logEntry + Environment.NewLine, Encoding.UTF8);

}

catch (Exception ex)

{

// En caso de error de escritura, se podría intentar escribir a la consola

// o a un sistema de diagnóstico alternativo.

Console.WriteLine($"Error al escribir en el archivo de log '{\_logFilePath}': {ex.Message}");

Console.WriteLine($"Mensaje original: {message}");

}

}

public void Info(string message) => WriteLog("INFO", message);

public void Debug(string message) => WriteLog("DEBUG", message);

public void Warn(string message) => WriteLog("WARN", message);

public void Error(string message) => WriteLog("ERROR", message);

public void Fatal(string message) => WriteLog("FATAL", message);

public void InfoFormat(string message, params object[] args) => WriteLog("INFO", string.Format(message, args));

public void DebugFormat(string message, params object[] args) => WriteLog("DEBUG", string.Format(message, args));

public void WarnFormat(string message, params object[] args) => WriteLog("WARN", string.Format(message, args));

public void ErrorFormat(string message, params object[] args) => WriteLog("ERROR", string.Format(message, args));

public void FatalFormat(string message, params object[] args) => WriteLog("FATAL", string.Format(message, args));

}

}

## 3. Clase LoggerAdapter.cs

La clase LoggerAdapter se usa para implementar IAppLogger (asumiendo que IAppLogger es lo mismo que ILogger en este contexto, o una interfaz base) y delega las llamadas a una instancia interna de Logger. Esto es útil si en el futuro se quiere cambiar la implementación subyacente del logger sin modificar el código que ya lo usa.

using System;

namespace Solutions.Utility.AppLogger

{

// Asumimos que IAppLogger es ILogger. Si fuera una interfaz diferente, debería definirse.

// Para este ejemplo, IAppLogger es sinónimo de ILogger.

public interface IAppLogger : ILogger { }

public class LoggerAdapter : IAppLogger

{

private readonly Logger \_logger;

public LoggerAdapter()

{

\_logger = new Logger();

}

// Delegación de todos los métodos a la instancia interna de Logger

public void AddFile(string file) => \_logger.AddFile(file);

public void AddPath(string path) => \_logger.AddPath(path);

public void DateFormatLog(string format) => \_logger.DateFormatLog(format);

public void Info(string message) => \_logger.Info(message);

public void Debug(string message) => \_logger.Debug(message);

public void Warn(string message) => \_logger.Warn(message);

public void Error(string message) => \_logger.Error(message);

public void Fatal(string message) => \_logger.Fatal(message);

public void InfoFormat(string message, params object[] args) => \_logger.InfoFormat(message, args);

public void DebugFormat(string message, params object[] args) => \_logger.DebugFormat(message, args);

public void WarnFormat(string message, params object[] args) => \_logger.WarnFormat(message, args);

public void ErrorFormat(string message, params object[] args) => \_logger.ErrorFormat(message, args);

public void FatalFormat(string message, params object[] args) => \_logger.FatalFormat(message, args);

}

}

## 4. Clase de Extensión para IServiceCollection (por ejemplo, LoggerServiceExtensions.cs)

Esta clase es crucial para la integración del logger con el sistema de inyección de dependencias de .NET.

using Microsoft.Extensions.DependencyInjection;

using System;

namespace Solutions.Utility.AppLogger

{

public static class LoggerServiceExtensions

{

public static IServiceCollection AddLogger(this IServiceCollection services)

{

// Aquí usamos LoggerAdapter porque es la clase que se instanciará.

// Si IAppLogger fuera distinto de ILogger, se usaría IAppLogger como tipo de retorno.

IAppLogger logging = new LoggerAdapter();

// Configuración de la ruta y el nombre del archivo de log

logging.AddPath(@"C:\Logs"); // Se recomienda usar la ruta completa para evitar ambigüedades.

logging.AddFile($"LOG\_Main.Service.Api\_{DateTime.Now:yyyyMMdd}.log"); // Formato de fecha simplificado

logging.DateFormatLog("yyyy-MM-dd HH:mm:ss.fff"); // Configura el formato de la fecha dentro del log

// Registra la instancia de IAppLogger como un Singleton.

// La misma instancia se usará en toda la aplicación.

services.AddSingleton(logging);

return services;

}

}

}

## 5. Ejemplo de Uso en una Aplicación (por ejemplo, Program.cs para una Consola o Startup.cs para ASP.NET Core)

Aquí te muestro cómo configurar y usar el logger en una aplicación de consola para simplificar el ejemplo. En un proyecto ASP.NET Core, la configuración iría en Startup.cs y la inyección en los controladores o servicios.

using Microsoft.Extensions.DependencyInjection;

using System;

using Solutions.Utility.AppLogger; // Asegúrate de incluir el namespace

namespace MyApp

{

public class Program

{

public static void Main(string[] args)

{

// 1. Configurar la inyección de dependencias

var serviceCollection = new ServiceCollection();

serviceCollection.AddLogger(); // Llama a tu método de extensión para agregar el logger

// 2. Construir el proveedor de servicios

var serviceProvider = serviceCollection.BuildServiceProvider();

// 3. Obtener la instancia del logger

// Asegúrate de que IAppLogger esté definida o usa ILogger si es la interfaz directa.

// Para este ejemplo, IAppLogger es ILogger.

var logger = serviceProvider.GetService<IAppLogger>();

if (logger == null)

{

Console.WriteLine("Error: El logger no pudo ser resuelto desde el contenedor de servicios.");

return;

}

// 4. Usar el logger

logger.Info("La aplicación ha iniciado correctamente.");

logger.DebugFormat("Procesando {0} elementos...", 10);

try

{

int a = 10;

int b = 0;

int result = a / b; // Esto causará una excepción

}

catch (DivideByZeroException ex)

{

logger.ErrorFormat("Se produjo un error crítico: {0}", ex.Message);

logger.Warn("Por favor, revise la configuración de la división.");

logger.Fatal("La aplicación no puede continuar debido a un error irrecuperable.");

}

finally

{

logger.Info("La aplicación ha finalizado.");

}

Console.WriteLine("Revisa el archivo de log en C:\\Logs para ver los mensajes.");

Console.ReadKey();

}

}

}

# Cómo Usar Este Documento Funcional:

1. **Crea un Proyecto de Biblioteca de Clases en C#** llamado Solutions.Utility.AppLogger.
2. **Agrega los Archivos:**
   * Crea ILogger.cs y copia el contenido del primer bloque de código.
   * Crea Logger.cs y copia el contenido del segundo bloque de código.
   * Crea LoggerAdapter.cs y copia el contenido del tercer bloque de código.
   * Crea LoggerServiceExtensions.cs y copia el contenido del cuarto bloque de código.
3. **Agrega las Dependencias de NuGet:**
   * Para LoggerServiceExtensions.cs, necesitarás el paquete Microsoft.Extensions.DependencyInjection. Instálalo en tu proyecto de biblioteca.
4. **Crea un Proyecto de Consola (o ASP.NET Core)** que haga referencia a tu proyecto Solutions.Utility.AppLogger.
5. **Agrega las Dependencias de NuGet:**
   * En tu proyecto de consola (o ASP.NET Core), también necesitarás Microsoft.Extensions.DependencyInjection.
6. **Copia el Código de Ejemplo** del Program.cs (o Startup.cs para ASP.NET Core) en tu proyecto de aplicación.
7. **Ejecuta la Aplicación.** Deberías ver un archivo de log creado en C:\Logs con los mensajes registrados.

Este conjunto de archivos te proporciona una implementación funcional del sistema de logging descrito en tu especificación. Si quieres hacer pruebas en diferentes entornos (como Linux o macOS), asegúrate de ajustar la ruta del log (C:\\Logs) a una ruta válida en esos sistemas (por ejemplo, /var/log/YourApp).