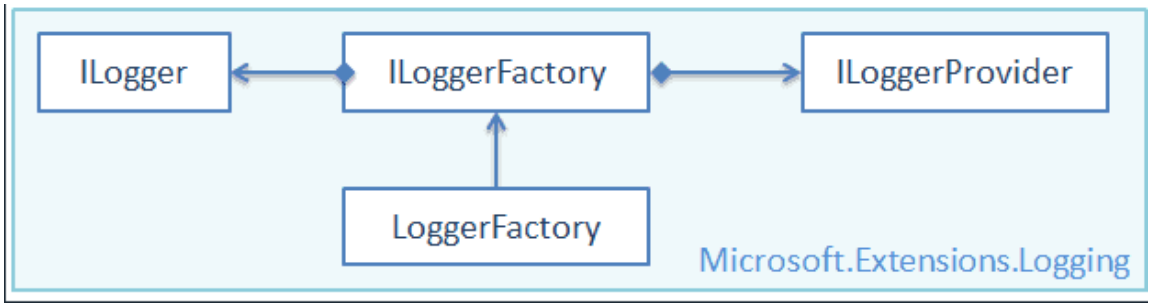
5. Logging in ASP.NET Core

- **Logging API**

The logging API in Microsoft.Extensions.Logging works on the .NET Core based applications whether it is ASP.NET Core or EF Core. You just need to use the logging API with one or more logging providers to implement logging in any .NET Core based application.

Microsoft provides logging API as an extension in the wrapperMicrosoft.Extensions.Logging which comes as a NuGet package. Microsoft.Extensions.Logging includes the necessary classes and interfaces for logging. The most important are the ILogger, ILoggerFactory, ILoggerProvider interfaces and the LoggerFactory class. The following figure shows the relationship between logging classes.



- **ILogger Factory**

The ILoggerFactory is the factory interface for creating an appropriate ILogger type instance and also for adding the ILoggerProvider instance.

```
public interface ILoggerFactory : IDisposable
{
    ILogger CreateLogger(string categoryName);
    void AddProvider(ILoggerProvider provider);
}
```

The Logging API includes the built-in LoggerFactory class that implements the ILoggerFactory interface. We can use it to add an instance of type ILoggerProvider and to retrieve the ILogger instance for the specified category.

- **ILogger Provider**

  The ILoggerProvider manages and creates an appropriate logger, specified by the logging category.
  We can create our own logging provider by implementing the ILoggerProvider interface.

```
public interface ILoggerProvider : IDisposable
{
    ILogger CreateLogger(string categoryName);
}
```

- **ILogger**

The ILogger interface includes methods for logging to the underlying storage. There are many extension methods which make logging easy.

The ILogger interface includes methods for logging to the underlying storage. There are many extension methods which make logging easy.

```
public interface ILogger
{
    void Log<TState>(LogLevel logLevel, EventId eventId, TState state, Exception excep
    bool IsEnabled(LogLevel logLevel);
    IDisposable BeginScope<TState>(TState state);
}
```

- **Configure logging**

Logging configuration is commonly provided by the Logging section of appsettings.{ENVIRONMENT}.json files, where the {ENVIRONMENT} placeholder is the environment. The following appsettings.Development.json file is generated by the ASP.NET Core web app templates:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  }
}
```

- **Logging Providers**

A logging provider displays or stores logs to a particular medium such as a console, a debugging event, an event log, a trace listener, and others. Microsoft provides various logging providers as NuGet packages.

The following table lists important logging providers.

Logging Provider's NuGet Package	Output Target
Microsoft.Extensions.Logging.Console	Console
Microsoft.Extensions.Logging.AzureAppServices	Azure App Services 'Diagnostics logs' and 'Log stream' features
Microsoft.Extensions.Logging.Debug	Debugger Monitor
Microsoft.Extensions.Logging.EventLog	Windows Event Log
Microsoft.Extensions.Logging.EventSource	EventSource/EventListener
Microsoft.Extensions.Logging.TraceSource	Trace Listener

Microsoft has also collaborated with various logging framework teams (including third parties like NLog, Serilog, Loggr, Log4Net, and others) to extend the list of providers compatible with Microsoft.Extensions.Logging.

The following are some third-party logging providers:

Loggr	Provider for the Logger service
NLog	Provider for the NLog library
Serilog	Provider for the Serilog library

Logging providers store logs, except for the Console provider which displays logs. For example, the Azure Application Insights provider stores logs in Azure Application Insights. Multiple providers can be enabled.

The default ASP.NET Core web app templates call WebApplication.CreateBuilder, which adds the following logging providers:

- Console
- Debug
- EventSource
- EventLog: (Windows only)

- **Console**

The Console provider logs output to the console. For more information on viewing Console logs in development.

```
.ConfigureLogging(logging =>
{
    logging.AddJsonConsole(options =>
    {
        options.TimestampFormat = "[HH:mm:ss] ";
    });
```

```
{
  "Logging": {
    "Console": {
      "LogLevel": {
        "Default": "Information",
        "Microsoft.AspNetCore": "Warning"
      },
      "FormatterName": "json",
      "FormatterOptions": {
        "TimestampFormat": "[HH:mm:ss] ",
        "JsonWriterOptions": {
          "Indented": true
        }
      }
    }
  },
  "AllowedHosts": "*"
}
```

- **Debug**

  The Debug provider writes log output by using the System.Diagnostics.Debug class. Calls to System.Diagnostics.Debug.WriteLine write to the Debug provider.

- **Event Source**

  The EventSource provider writes to a cross-platform event source with the name Microsoft-Extensions-Logging. On Windows, the provider uses ETW.

  Event Tracing for Windows (ETW) provides application programmers the ability to start and stop event tracing sessions, instrument an application to provide trace events, and consume trace events. Trace events contain an event header and provider-defined data that describes the current state of an application or

operation. You can use the events to debug an application and perform capacity and performance analysis.

- **Event Log**

  The EventLog provider sends log output to the Windows Event Log. Unlike the other providers, the EventLog provider does *not* inherit the default non-provider settings. If EventLog log settings aren't specified, they default to LogLevel.Warning.

  To log events lower than LogLevel.Warning, explicitly set the log level. The following example sets the Event Log default log level to LogLevel.Information

```
"Logging": {
  "EventLog": {
    "LogLevel": {
      "Default": "Information"
    }
  }
}
```

```
    logging.AddEventSourceLogger();

    logging.AddEventLog(options =>
    {
        options.SourceName = sourceName;
        options.LogName = "Application";
        options.Filter = (category, level) =>
        {
            return level >= LogLevel.Warning;
        };
    });
    logging.SetMinimumLevel(LogLevel.Trace);
})
```

```
if (!EventLog.SourceExists(sourceName))
{
    EventLog.CreateEventSource(new EventSourceCreationData(sourceName, "Application"));
}
```

- **Log Level**

LogLevel	Value	Method	Description
Trace	0	LogTrace	Contain the most detailed messages. These messages may contain sensitive app data. These messages are disabled by default and should ***not*** be enabled in production.
Debug	1	LogDebug	For debugging and development. Use with caution in production due to the high volume.
Information	2	LogInformation	Tracks the general flow of the app. May have long-term value.
Warning	3	LogWarning	For abnormal or unexpected events. Typically includes errors or conditions that don't cause the app to fail.
Error	4	LogError	For errors and exceptions that cannot be handled. These messages indicate a failure in the current operation or request, not an app-wide failure.
Critical	5	LogCritical	For failures that require immediate attention. Examples: data loss scenarios, out of disk space.
None	6		Specifies that a logging category shouldn't write messages.