# Built-in Logging

.NET CORE

*When things go wrong in production, you must gather information. An effective logging system is essential to ensure that you have some idea of where to look for the cause if an error.*

https://visualstudiomagazine.com/articles/2019/03/22/logging-in-net-core.aspx

# Logging - Overview

https://docs.microsoft.com/en-us/dotnet/core/diagnostics/logging-tracing

---

'***Logging***' involves instrumenting an application to write output (to a file or to console) under certain situations. A logging ***Provider*** stores logs in a specified place.

***Logging*** is useful in situations where the debugger falls short, such as:

- Issues occurring over long periods of time.
- When analysis is required to understand complex systems after a crash. A debugger tends to modify program behavior.
- When attaching a debugger causes timeout failures.
- When programs need to always be recording. ***Logging*** is designed for low overhead, so they require very little from the system.

# Logging – Performance Considerations

https://docs.microsoft.com/en-us/dotnet/core/diagnostics/logging-tracing#performance-considerations
https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging/?view=aspnetcore-3.1#no-asynchronous-logger-methods

---

***Logging*** should be so fast that it isn't worth the performance cost of ***asynchronous*** code.

It is recommended that you:

- Avoid lots of ***logging*** when no one is listening.
- Avoid constructing costly ***logging*** messages by checking if ***logging*** is enabled first.
- Only log what's useful.
- Defer fancy formatting to the analysis stage.

# Logging – .NET Print-Style APIs

https://docs.microsoft.com/en-us/dotnet/core/diagnostics/logging-tracing#print-style-apis

The choice of which 'print-style' API to use is up to you.

| System.Console | System.Diagnostics.Trace | System.Diagnostics.Debug |
|---|---|---|
| • Always enabled and always writes to the console.<br>• Useful for information that your customer may need to see in the release.<br>• Because it's the simplest approach, it's often used for ad-hoc temporary debugging.<br>• This debug code is often never checked in to source control. | • Only enabled when ***TRACE*** is defined.<br>• Writes to attached Listeners, by default the ***DefaultTraceListener***.<br>• Use this API when creating logs that will be enabled in most builds. | • Only enabled when ***DEBUG*** is defined.<br>• Writes to an attached debugger.<br>• On *nix writes to ***stderr*** if ***COMPlus_DebugWriteToStdErr*** is set.<br>• Use this API when creating logs that will be enabled only in debug builds. |

# Logging - Events

https://docs.microsoft.com/en-us/dotnet/core/diagnostics/logging-tracing#print-style-apis

Rather than logging simple strings these APIs log event objects.
They can be utilized with using System.Diagnostics.*

| .Tracing.EventSource | .DiagnosticSource | .Activity | .EventLog |
|---|---|---|---|
| • ***EventSource*** is the primary root ***.NET Core*** tracing API.<br>• Available in all ***.NET Standard*** versions.<br>• Only allows tracing serializable objects.<br>• Writes to the attached event listeners.<br>• ***.NET Core*** provides listeners for:<br>  • ***.NET Core's*** EventPipe on all platforms<br>  • Event Tracing for Windows (ETW)<br>  • LTTng tracing framework for Linux | • Included in ***.NET Core*** and as a ***NuGet*** package for ***.NET Framework***.<br>• Allows in-process tracing of non-serializable objects.<br>• Includes a bridge to allow selected fields of logged objects to be written to an ***EventSource***. | • Provides a definitive way to identify log messages resulting from a specific activity or transaction.<br>• This object can be used to correlate logs across different services. | • Windows only.<br>• Writes messages to the ***Windows Event Log***.<br>• System administrators expect fatal application error messages to appear in the ***Windows Event Log***. |

# ILogger

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging/?view=aspnetcore-3.1#built-in-logging-providers
https://www.blinkingcaret.com/2018/02/14/net-core-console-logging/
https://docs.microsoft.com/en-us/aspnet/core/fundamentals/host/generic-host?view=aspnetcore-3.1

---

- ***.NET Core*** has a built-in ***logging*** API (***ILogger***) that works with a variety of built-in and third-party logging providers.
- ***ILogger*** can be used with ***HostBuilder***.
- ***ILogger*** provides multiple ways to display or store logs. Logs can be sent to multiple destinations by adding multiple providers.
- ***Logging*** code for apps without ***Generic Host*** differs in the way providers are added and ***loggers*** are created.

# ILogger Built-in Providers

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging/?view=aspnetcore-3.1#built-in-logging-providers

| Provider | Description |
|---|---|
| Console | The ***Microsoft.Extensions.Logging.Console*** package sends log output to the console. |
| Debug | The ***Microsoft.Extensions.Logging.Debug*** package writes log output by using the ***System.Diagnostics.Debug*** class. |
| Event Source | The ***Microsoft.Extensions.Logging.EventSource*** package writes to an Event Source cross-platform with the name ***Microsoft-Extensions-Logging***. ***.EventSource*** is added automatically when ***CreateDefaultBuilder*** is called to build the host. |
| Windows EventLog | The ***Microsoft.Extensions.Logging.EventLog*** package sends log output to the Windows Event Log. |
| TraceSource | The ***Microsoft.Extensions.Logging.TraceSource*** provider package uses the TraceSource libraries and providers. |
| Azure App Service | The ***Microsoft.Extensions.Logging.AzureAppServices*** provider package writes logs to text files in an Azure App Service app's file system and to blob storage in an Azure Storage account. |
| Azure Application Insights trace logging | The ***Microsoft.Extensions.Logging.ApplicationInsights*** package writes logs to Azure Application Insights. |

# ILogger Configuration

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging/?view=aspnetcore-3.1#configuration

Logging configuration is done in various files and in various languages..

- File formats (INI, JSON, and XML).
- Command-line arguments.
- Environment variables.
- In-memory .NET objects.
- The unencrypted Secret Manager storage.
- An encrypted user store, such as Azure Key Vault.
- Custom providers (installed or created).
- Most commonly, ***appsettings.json*** is used to configure built-in logging

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    },
    "Console":
    {
      "IncludeScopes": true
    }
  }
}
```

# ILogger - Log Levels

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging/?view=aspnetcore-3.1#log-level

- The ***Logging*** property in ***appsettings.json*** can have ***LogLevel*** and ***log provider*** properties.
- The ***LogLevel*** property under ***Logging*** specifies the minimum level to log for selected categories.
  - (Optional) ***LogLevel*** under a provider specifies levels to log for that provider.
  - levels specified in ***Logging.{providername}.LogLevel*** override anything set in ***Logging.LogLevel.***

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    },
    "Console":
    {
      "IncludeScopes": true
    }
  }
}
```

# ILogger – Log Levels

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging/?view=aspnetcore-3.1#log-level

| Log Level | Description |
|---|---|
| Trace = 0 | For information that's typically valuable only for debugging. These messages may contain sensitive application data and so shouldn't be enabled in a production environment. Disabled by default. |
| Debug = 1 | For information that may be useful in development and debugging. Example: Entering method Configure with flag set to true. Enable Debug level logs in production only when troubleshooting, due to the high volume of logs. |
| Information = 2 | For tracking the general flow of the app. These logs typically have some long-term value. Example: Request received for path /api/todo |
| Warning = 3 | For abnormal or unexpected events in the app flow. These may include errors or other conditions that don't cause the app to stop but might need to be investigated. Handled exceptions are a common place to use the Warning log level. Example: FileNotFoundException for file quotes.txt. |
| Error = 4 | For errors and exceptions that cannot be handled. These messages indicate a failure in the current activity or operation (such as the current HTTP request), not an app-wide failure. Example log message: Cannot insert record due to duplicate key violation. |
| Critical = 5 | For failures that require immediate attention. Examples: data loss scenarios, out of disk space. |

# ILogger – Log Levels

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging/?view=aspnetcore-3.1#log-level

Use the ***log level*** to control how much log output is written to your logging destination.

Logging Rules of thumb.

- In production - Logging at the ***Trace*** through ***Information*** levels produces a high-volume of detailed log messages. Log ***Trace*** through ***Information*** level messages to a high-volume, low-cost data store.
- Logging at ***Warning*** through ***Critical*** levels typically produces fewer, smaller log messages. Therefore, costs and storage limits usually aren't a concern, which results in greater flexibility of data store choice.
- During development - Log ***Warning*** through ***Critical*** messages to the ***console***. Add ***Trace*** through ***Information*** messages when troubleshooting.

```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GetItem, "Getting item {Id}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({Id}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```

# ILogger – Log Filtering

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging/?view=aspnetcore-3.1#log-filtering

Specify a minimum log level for a ***provider*** and ***category*** or for all ***providers*** or all ***categories***.

Any logs below the minimum level aren't passed to that provider, so they don't get displayed. If the default value is ***Information***, ***Trace*** and ***Debug*** logs are ignored.

Each provider defines an alias that can be used during configuration. For built-in providers, use the following aliases:

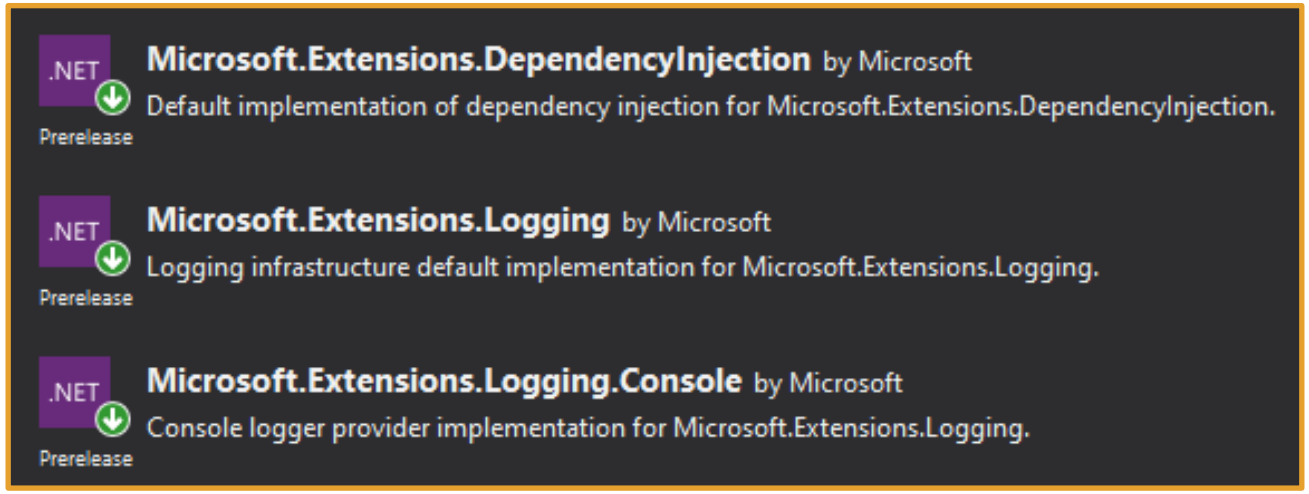
- Console, Debug, EventSource, EventLog, TraceSource, AzureAppServicesFile, AzureAppServicesBlob, ApplicationInsights

```
{
  "Logging": {
    "Debug": {
      "LogLevel": {
        "Default": "Information"
      }
    },
    "Console": {
      "IncludeScopes": false,
      "LogLevel": {
        "Microsoft.AspNetCore.Mvc.Razor.Internal": "Warning",
        "Microsoft.AspNetCore.Mvc.Razor.Razor": "Debug",
        "Microsoft.AspNetCore.Mvc.Razor": "Error",
        "Default": "Information"
      }
    },
    "LogLevel": {
      "Default": "Debug"
    }
  }
}
```

# Logging in a Console App Step-by-Step

https://www.blinkingcaret.com/2018/02/14/net-core-console-logging/

1. Add three Nuget Packages.




2. Add Logging Service in Program.cs and configure logging.

```
var services = new ServiceCollection();
ConfigureServices(services);

using (ServiceProvider serviceProvider = services.BuildServiceProvider())
{
    GamePlay game = serviceProvider.GetService<GamePlay>();

    game.GetPlayersName();
    game.RunGame();
    game.PrintResults();
}
```

# Logging in a Console App Step-by-Step

https://www.blinkingcaret.com/2018/02/14/net-core-console-logging/

3. Add ConfigureServices() below Main().

```
private static void ConfigureServices(ServiceCollection services)
{
    services.AddLogging((configure) =>
    {
        configure.ClearProviders();
        configure.AddConsole();
        configure.SetMinimumLevel(LogLevel.Trace);
    })
    .AddTransient<GamePlay>();
}
```

4. Inject the Logging Service into the constructor of the class you want to log in.

```
private readonly ILogger _logger;
public GamePlay(ILogger<GamePlay> logger)
{
    _logger = logger;
}
```

# Logging in a Console App Step-by-Step

https://www.blinkingcaret.com/2018/02/14/net-core-console-logging/

5. Use the ***_logger*** to log wherever you want to log inside the class.

```
public void GetPlayersName()
{
    _logger.LogInformation("LogInformation = Hello. My
      name is Log LobInformation");
    _logger.LogWarning("LogWarning = Now I'm Loggy
      McLoggerton");
    _logger.LogCritical("LogCritical = As of now, I'm
      Scrog McLog");
    _logger.LogDebug("Log Debug");
    _logger.LogError("LogError");
    _logger.LogTrace("Log Trace = Tracing my way back
      home.");
```

6. View the ***logging levels*** shown to the console. ***.logDebug()*** and ***.logTrace()*** are not printed to console by default

```
Enter Player1 Name:
info: RPS_Game.GamePlay[0]
      LogInformation = Hello. My name is Log LobInformation
warn: RPS_Game.GamePlay[0]
      LogWarning = Now I'm Loggy McLoggerton
crit: RPS_Game.GamePlay[0]
      LogCritical = As of now, I'm Scrog McLog
fail: RPS_Game.GamePlay[0]
      LogError
```

# ILogger in a Web App - Step-by-Step

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging/?view=aspnetcore-3.1
https://www.youtube.com/watch?v=oXNslgIXIbQ

---

1. To add a provider in an app that uses Generic Host,
2. Add ***Microsoft.Extensions.Logging*** NuGet Package
3. add using Microsoft.Extensions.Logging at the top of your ***Program.cs.***
4. Under Host.CreateDefaultBuilder(args), clear out default logging settings with ***logging.ClearProviders();.***
5. Call the logging.Add{provider name} extension method in ***Program.cs***. (Ex. logging.AddConsole();, logging.AddDebug();, etc.)
6. Add using Microsoft.Extensions.Logging; to any class where you will be logging.
7. Use ***Dependency Injection*** to inject an instance of an ILogger<ContainingClassName> object into each class where you will use logging.
8. Use the different logging levels to note different events in your applications processes.

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureLogging(logging =>
        {
            logging.ClearProviders();
            logging.AddConsole();
        })
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

```
public class AboutModel : PageModel
{
    private readonly ILogger _logger;

    public AboutModel(ILogger<AboutModel> logger)
    {
        _logger = logger;
    }
```

# Logging in a Console App Step-by-Step

https://www.blinkingcaret.com/2018/02/14/net-core-console-logging/

---

A good source on DI - https://andrewlock.net/using-dependency-injection-in-a-net-core-console-application/

Good source to talk about App Configuration - https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration talks about how ***appsettings.json*** is called.. Then userSecrets.json, etc

https://docs.microsoft.com/en-us/aspnet/core/security/app-secrets?view=aspnetcore-3.1&tabs=windows#enable-secret-storag

# Tutorial with LoggerFactory

https://docs.microsoft.com/en-us/ef/core/miscellaneous/logging?tabs=v3

---

Work in Progress