Logging with enums in .NET Core

Vidar Kongsli Follow
Jun 25, 2018 · 3 min read

In this blog post, I extend the built-in logging framework with the use of enums as log events, building on the out-of-the-box log event concept.

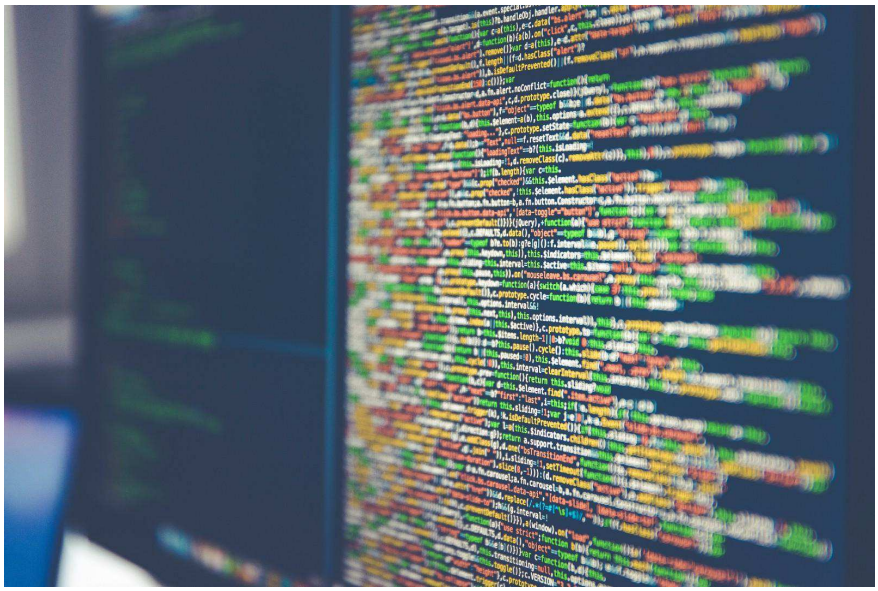

Photo by Markus Spiske on Unsplash

A brief introduction

With the `Microsoft.Extensions.Logging` package, flexible and powerful logging concepts are available. This is an integral part of dotnet core. The new logging framework has integrated concepts like log categories, familiar to users of the Log4j, Log4net and NLog logging libraries, and structured (a.k.a. semantic) logging, familiar to users of Serilog.

Microsoft has a nice introduction to the logging concepts here, and here is a short snippet that give you an idea of the standard functionality in ASP.NET core:

```
public class HomeController : Controller
{
    private readonly ILogger<HomeController> _logger;

    public HomeController(ILogger<HomeController> logger)
    {
        _logger = logger;
    }
    public IActionResult Index()
    {
        _logger.LogInformation("Index read {user}",
```

An implementation if `ILogger<T>` is automatically injected into the constructor by the built-in dependency injection container. This type also has overloaded methods that take an `EventId` as the first parameter, for example `void LogInformation(EventId eventId, string message, params object[] args)` Basically, you pass in any integer as the first parameter as the `EventId` type has an implicit operator converting an `int` to `EventId` : `public static implicit operator EventId(int)` . So, the typical example is that you create a class containing constants that represent your application events:

```
public class LoggingEvents
{
    public const int GenerateItems = 1000;
    public const int ListItems = 1001;
    public const int GetItem = 1002;
    public const int InsertItem = 1003;
    public const int UpdateItem = 1004;
    public const int DeleteItem = 1005;
```

And then you use this like so:

```
public IActionResult Index()
{
    _logger.LogInformation(LoggingEvents.ListItems, "Index r
        Request.HttpContext.User);
```

Using enums as event ids

So, the first improvement that I suggest, is to use enums instead of having a static class containing a list of integer constants containing my event ids. What I would like to have, a definition of event ids like this:

```
public enum LoggingEvents
{
    GenerateItems = 1000,
    ListItems,
    GetItem,
    InsertItem,
    UpdateItem,
    DeleteItem,
```

So, let's have a first shot at an overloaded extension method that will implement this:

```
public static class LoggerExtensions
{
    public static void LogInformation<T>(this ILogger<T> @th
        string message, params object[] args)
    {
        var i = (int)Convert.ChangeType(e, e.GetTypeCode());
        @this.LogInformation(i, message, args);
```

Notice that we use the `Enum` type for the parameter, which is the ancestor type for all enums in C#. Then we do some magic to get the integer representation of the passed enum instance. Then, the log statement looks similar to the one above, but uses an enum instead of an integer: `_logger.LogInformation(LoggingEvents.ListItems, "Index read {user}", Request.HttpContest.User);`

Let's try to improve this a bit by leveraging the fact that we are using enums. As I mentioned before, the built in logging features so-called semantic or structured logging. So let's add some more semantics to our extension method above:

```
public static void LogInformation<T>(this ILogger<T> @this,
    string message, params object[] args)
{
    var i = (int)Convert.ChangeType(e, e.GetTypeCode());
    var eventName =  Enum.GetName(e.GetType(), e);
    var eventScope = e.GetType().Name;
    var parameters = args.Append(eventScope).Append(eventNa
```

There's a lot of stuff going on here, but let's try to sort it out:

- The `eventName` variable would hold the enum name, like *ListItems*.
- The `eventScope` variable will be the name of the enum type, like *LoggingEvents*.
- The variables above are added to the log message parameters, so that they automatically become a part of the log message structure and are available for filtering, sorting, etc. when making sense of the logs.

Now, it would be easier to comprehend the event ids given in the logs, by having the enum type names and enum names for better readability. This approach is highly influenced by an idea by an ex-colleague of mine, Thomas Svensen, which you can read about here.

Using several enum types

With having the name and scope parameters attached to the log messages, it is feasible to structure the log events using several enum types. You could, for instance, do something like this:

```
public enum CustomerEvents
{
    CustomerCreated = 1000,
    CustomerRead,
    CustomerChanged,
    CustomerDeleted
}

public enum OrderEvents
{
```

The only thing you would need to make sure, is that the integer values of the different enum types do not overlap, as that would mess up the event ids.

Centralizing log levels

One thing that always puzzles me when coding, is to figure out the correct log level (*information, warning, error, critical*, etc.). To me, it is very difficult to comprehend this while writing code. So, what I would like to do, is to move the log level concern out of the application code. Instead of having it sprinkled around the code, let's make a map that maps the events to log levels:

```
private static readonly Dictionary<Enum, LogLevel> LogLevel
{
    { AppEvents.GenerateItems, LogLevel.Information },
    { AppEvents.ListItems, LogLevel.Information },
    { AppEvents.GetItem, LogLevel.Information },
    { AppEvents.InsertItem, LogLevel.Information },
    { AppEvents.UpdateItem, LogLevel.Information },
    { AppEvents.DeleteItem, LogLevel.Information },
```

Now we don't have to implement overloads for `LogInformation`, `LogError`, `LogWarning`, and so on. We need only one overloaded method, `Emit`:

```
private const LogLevel DefaultLogLevel = LogLevel.None;

public static void Emit<T>(this ILogger<T> @this, Enum e, s
    params object[] args)
{
    var i = (int)Convert.ChangeType(e, e.GetTypeCode());
    var eventName = Enum.GetName(e.GetType(), e);
    var eventScope = e.GetType().Name;
    var level = LogLevelMap.ContainsKey(e) ? LogLevelMap[e]
```

Then we can write log statements like these:

```
var id = 1;
_logger.Emit(LoggingEvents.GetItem, "Item({id})", id);
_logger.Emit(LoggingEvents.ListItems);
```