EF: Logging

Here's some notes on how we implement logging in EF.

We've currently implemented it in our dynamic domain dbcontext, DynamicDbContext.

To make it work, we just have to create an instance of DbContextLoggingConfig, and pass it to our top-level dbcontext constructor.

> NOTE: Our logging implementation is to use Nlog as a log sink.
> But, this page contains notes on how to adapt the logging boilerplate to other backend types.

DbContextLoggingConfig

Our logging implementation requires an instance of this class to define the EF logging behavior.

It is passed to the dbcontext constructor, and remains through the lifecycle of the dbcontext.

> NOTE: It is possible to change logging behavior at runtime, by creating a new instance of this, that is passed to newly generated dbcontext instances.
> But, we will leave that as an advanced topic.

Here are some notes on how to setup the config.

Enabled

To run a dbcontext without logging, you don't actually have to pass an instance of DbContextLoggingConfig to the dbcontext constructor.
But if you do want to pass one in, and still have logging disabled, just set Enabled = False;

IncludeSensitiveData

If you set IncludeSensitiveData to True, then EF log events will include things, such as passwords and connection strings.
Specifically, setting this makes a call to options.EnableSensitiveDataLogging().

Level

This defines the logging level for EF logging.
Set the Level to the desired logging level for EF log-able events.

> NOTE: Your process logging has to meet or exceed the EF logging level, in order to see the higher-diagnostic messages from EF.
> This is because we are still using NLog as out logging output, and we translate the EF log message levels to NLog levels, to determine what level to log something at.

EventCategories

This is a bit-wise enumeration of EF events that can be logged.

We've grouped events into this enum: eEFLoggingEvents.

They are:

- Commands - Add this to capture high-level execution metrics and errors.
- Connection Life Cycle - Includes these Logging to diagnose connection pool issues or intermittent DB problems.
- Transactions - These are for debugging partial failures, deadlocks, rollback scenarios, and long-running transactions.
- Save Changes - These are useful to detect persistence-level exceptions.
- Query Warnings - These are useful to detect unordered paging, client-side evaluation risks, lazy loading mistakes.

> Some of these categories create tons of log messages and performance penalties.
> So, use care when enabling them in production.

To define the log event categories, you do an OR (||) of the desired categories.

Like this:

```
config.EventCategories = eEFLoggingEvents.Commands | eEFLoggingEvents.ConnectionLifeCycle;
```

UseEFLogClass

Normally, this will be False, so the running process will use the global logger.

But if you want a dedicated NLog logger, set this to true.

When true, a logger, named "EF" will be used.

>

> NOTE: Your process startup should provide any setup required for this named logger, so that it will have the desired formatting and target.

LoggerOptions

This property doesn't need adjustment, for our current implementation (using a custom log format).
So, it can be left alone.

This is because we accept the EventData object from EF, and apply our own formatting to the log message.

But if you do decide to accept pre-formatted log strings from EF, then this property would define some aspects of it, such as timestamp format, and line-wrap.

To apply options to the property, it is best to use the predefined option flags from the DbContextLoggerOptionsPresets class.

Sink

Normally, this property can be left alone, since our current implementation is to send logs to Nlog.

But if you do want logs to go elsewhere, assign a callback delegate, to this property that will accept and record formatted log messages from EF.

EFLogFilterBuilder

This helper class contains methods that are used by the logging boilerplate of the dbcontext.

> NOTE: There is no need to directly call methods of this class.
> But, we will explain what it does.

This class type's goal is to generate a filtering predicate, a callback method, that EF accepts as an argument in its options.LogTo() method.

The filter predicate tells EF what events are to be logged for the logging level.

You can evolve this class as you come across other EF event types to log, or have different category groupings to use.
Following its pattern, will generate the appropriate filter predicate methods that EF uses.

DbContext Logging Boilerplate

Here are notes on the boilerplate logic that is used in the dbcontext.

Constructor

The base constructor accepts an optional DbContextLoggingConfig instance.

If not given or is null, no EF logging will occur.

If you are adapting this for a new dbcontext constructor, be sure that the private logging config field (_logging) is always set by the constructor.
Here is an example of how to do it:

```
public DynamicDbContext([NotNullAttribute] DbContextOptions options, DbContextLoggingConfig? logging =
null) : base(options)
{
    // Accept any logging config that was passed in...
    _logging = logging ?? new DbContextLoggingConfig { Enabled = false };
}

public DynamicDbContext() : base()
{
    // Accept a default logging config...
    _logging = new DbContextLoggingConfig { Enabled = false };
}
```

Dispose Methods

Each of the Dispose method types (sync and async) need to release logging.
This is done, so that any Sink callback is cleared, preventing dangling references.

It can be done, like this:

```
override public void Dispose()
{
    if (!_disposed)
    {
        // Release logging...
        this.ReleaseLogging();

        _disposed = true;
    }
    base.Dispose();
```

```
}

/// <summary>
/// We override DisposeAsync() so we can unhook logging.
/// </summary>
override public async ValueTask DisposeAsync()
{
    if (!_disposed)
    {
        // Release logging...
        this.ReleaseLogging();

        _disposed = true;
    }
    await base.DisposeAsync();
}
```

OnConfiguring() Override

The OnConfiguring() override needs to grab the DbContextOptionsBuilder instance and setup logging with it.

This is done by simply calling into our logging boilerplate, like this:

```
protected override void OnConfiguring(DbContextOptionsBuilder options)
{
    // Do any logging setup...
    this.SetupLogging(options);
}
```

SetupLogging()

This method is called by OnConfiguring(), to stand up logging for the dbcontext.

It's purpose is to make a call to: options.LogTo(), and pass in a filter predicate and log callback.

> NOTE: options.LogTo() is our hook into the logging mechanism of EF.
> So, all of our logging logic works to call this method.

The options.LogTo() method is where we apply filtering and assign a logging callback method.

It accepts the logging filter predicate (created by EFLogFilterBuiler) to define what events to log.

And, it accepts the logging callback delegate, our EfEventInfeed() method, to perform the actual logging.

> NOTE: Specifically, the logging callback delegate is directly called by EF core, when a log-able event has occurred.
> The delegate's job is to do the actual logging.

There is no need to modify anything in this method, to change logging behavior.

If you are adapting logging to a new dbcontext type, just copy this method across.

ToNLogLevel()

This method is used to convert the Microsoft log levels to our NLog log levels.
It is called each time a log-able event triggers.

If you are adapting logging to a new dbcontext type, just copy this method across.

EfEventInfeed()

This is the callback method that hooks into EF, to receive log-able events.

This method performs the actual logging.

It will send logs to one of three places:

- If the Sink property (of the logging config) is set, logs will be diverted there.
- If the UseEFLogClass (of the logging config) is True, logs will be sent to the named NLog logger, "EF".
- Otherwise, logs will be sent to the normal NLog target.

If you are adapting logging to a new dbcontext type, just copy this method across.

If you are adapting a new logging backend, this is where you will start splicing in your logging backend, replacing the Nlog calls with yours.

FormatLogMessage()

This method is a bit of purposeful function indirection, that we use to reduce unnecessary formatting of messages that will not be logged.

> NOTE: If you review the EfEventInfeed() method, you will notice that we feed this FormatLogMessage() method directly into the nlog.Log() statement.
> That ensures that the FormatLogMessage() call is only executed IF the Nlog logging level is adequate.

When executed, this method composes the log message with a prepended context line that contains data from the log level and eventdata.

It returns the composed log string, for logging.

If you are adapting logging to a new dbcontext type, just copy this method across.

If you are wanting to change the shape and information of log messages, modify this method.

ReleaseLogging()

This method is used during Dispose(), to release any explicit log sink callback.

If you are adapting logging to a new dbcontext type, just copy this method across.

Revision #5
Created 18 February 2026 09:44:46 by glwhite
Updated 19 February 2026 06:43:43 by glwhite