Process Logging Behavior

Here’s my current conventions for logging in application processes.

Logging features, here are implemented by the Logging class in OGA.Common.Lib.

Logging Phases

We use three distinct logging phases for an application, each with a distinct purpose.

- Early Logging
- Startup Logging
- Normal Logging

Early Logging

Early logging is done during the process startup period when configuration is not yet loaded to determine what logging targets should be active.

As soon as practical, our starting process sets up logging to the console and a memory logger.

This allows us to capture those early diagnostic moments, before the process has been able to load enough configuration to know where to send log messages… like where the logging path is, what network log sink to use, etc…

In this logging phase, the default application logger instance is configured to send log messages to the console and to a memory logger.

All the logs collected by the memory logger will be played into the durable log store, in the next phase.

Startup Logging

This is the second logging phase of a running process.

This phase logs at a higher log level to capture log messages that may be useful to diagnose process startup issues.

It begins when the process has collected enough config information about logging targets and pathing, and has started logging to a durable log store.

This phase is temporary in nature, as it is only active for a short period at process startup… about 30 seconds.

After the startup logging timer expires, logging switches to the normal logging phase.

Normal Logging

This logging phase uses the regular logging level of the application, and sends logs to the durable log store.

This phase begins after the startup logging phase has ended, and the logging level switches to the normal process level.

Implementation

Here are details for logging setup, teardown and management in a .NET process.

Dependencies

Add reference to the following:

- OGA.SharedKernel
- OGA.Common
- NLog

Once references added, below are the blocks of code to add to your Program.cs.

For easier implementation, these are already included in the ProgramBase.cs of OGA.Common.Lib. So, updating your Program.cs to inherit from that ProgramBase, and calling its Consolidated_Main() will perform the below logging details.

Starting Early Logging

Kicking off early logging is straightforward to do.

It is currently done with this:

```
OGA.Common.Logging.Logging.Set_AppData_LogTarget("memory");
OGA.Common.Logging.Logging.Set_AppData_LogLevel(
        OGA.Common.Logging.Logging.Startup_Logging_Level.ToString());
if (OGA.Common.Logging.Logging.Start_Logging() < 0)
{
   Console.WriteLine("Failed to start logging.");
   return -1;
}
OGA.SharedKernel.Logging_Base.Logger_Ref?.Debug(
      "**********************************************************Initial Logging Started");
```

The above is part of the logic in Consolidated_Main.

It will set the log level to the startup level (Debug), and start logging to the memory target.

Startup Logging

Once the logic in Consolidated_Main has progressed enough, to determine process naming and retrieve enough configuration to know where to send logs, it will redirect logging to the normal logging targets, but at the startup log level.

This is done with the below calls:

```
OGA.Common.Logging.Logging.Set_AppData_LogTarget("file");
OGA.Common.Logging.Logging.Set_AppData_LogLevel(OGA.Common.Logging.Logging.Normal_Logging_Level.To
String());
if (OGA.Common.Logging.Logging.Start_Logging() < 0)
{
    // Failed to start logging.


    Console.WriteLine("Failed to start logging.");
    return -6;
}
// Tell the logger to arm its switchover delay, to normal logging...
OGA.Common.Logging.Logging.Enable_StartupLogSwitch();
OGA.SharedKernel.Logging_Base.Logger_Ref?.Debug(
        "**********************************************************Logging Started");
```

Normal Logging

The previous code block includes a call to enable startup logging switch:

```
OGA.Common.Logging.Logging.Enable_StartupLogSwitch();
```

Including this in the startup logging code block, will automatically transition from logging from the startup log level into the normal process log level… after the startup delay has expired.

Characterized Process vs Uncharacterized Process

It’s not uncommon for a compiled process binary to have more than one purpose in life.
And, it’s possible that multiple copies of the same binary will run on a host for different reasons.

This can be the case when you might run multiple copies of the same collector binary, each with a different station assignment (added at the command line).

So. For processes that run multiple copies of the same binary for different purposes, it can be useful for the logger to distinguish log filenames of each assigned station or duty of the process binary.

This allows us to have independent log files, for each station or duty of a given process binary.

To make this work, our logging logic looks at the “Is_ServiceSpecificProcess” flag on OGA.SharedKernel.Process.App_Data_v2.
If this flag is set, the Start_Logging() method call will include the servicename in the log filename.

This Service_Name property is also set in the same struct of:
OGA.SharedKernel.Process.App_Data_v2
To use this feature, apply whatever stationid or service context that you need to distinguish process logs to the Service_Name property and set the Is_ServiceSpecificProcess flag to true.

When done, your log file filenames will look like this:

<processname>-<servicename>_Log${date:format=yyyyMMdd}.log

Ex: DataCollector-Station123_Log20240311.log

Revision #1
Created 10 April 2025 05:14:15 by glwhite
Updated 10 April 2025 05:16:58 by glwhite