NuLog Documentation

Release 2.1.1

Ivan Pointer

Jul 06, 2017

User Documentation

Welcome to the NuLog documentation! These docs are under heavy development, and will be finished soon. Check back soon!

CHAPTER 1

Martin's First Law of Documentation

Produce no document unless its need is immediate and significant.

We feel that the best form of documentation for a software project, are well written business driven tests (*Architectural Principles*).

Head over to GitHub to look through the unit and integration tests for the project.

CHAPTER 2

What's Changed From Version 1

Please be aware: this is a ground-up rewrite of NuLog, so only the high-level concepts and architectural pattern have been kept. *This is a full-on breaking change, which is why the version number has been advanced to 2.X.*

Much of the functionality of version 1 has been rebuilt into version 2. However, the functionality has been simplified, and some features were deprecated, and not brought over. Here's a quick overview of what's changed:

- **Runtime Config Helpers** - The runtime config helpers haven't been built into version 2. This doesn't mean they won't be added in the future, but that they just haven't been needed yet.
- **MEF Deprecated** - MEF, while quite powerful, also leads to obfuscating functionality from the developer, which could lead to unexpected behavior (from the developer's perspective). The rewrite of the framework allows for easier extension, using the *Dependency Inversion Principle*, and MEF is no longer needed.
- **Static Meta Data Providers Deprecated** - The static meta data providers haven't been shown to be needed yet. They don't conflict with the new philosophy of NuLog, but are an example of complexity that hasn't been shown to be necessary yet, so they haven't been ported over - at least yet.
- **Configuration Changed from JSON to use standard XML-based config** - To better support standard CI processes, and to simplify the management of the configuration of the framework, the configuration has been moved away from a custom JSON based mechanism, to leverage the built-in configuration management. The configuration is now stored as a custom configuration section, right in the app/web config files.
- **Legacy Logging Extension Deprecated** - If neccessary, we can build this out as a separate package, but NuLog is a new approach to logging, and we encourage organizations to make the transition towards tag-based logging, as opposed to just emulating level-based logging, using a tag based framework.
- **Configuration Extenders Deprecated** - Another example of complexity that isn't yet necessary - the configuration extenders haven't been ported over. Again, if the need presents, we can build this out.
- **Console Targets Simplified** - The two console targets have been replaced with a single console target, which is somewhere between the two of the original, in terms of functionality.
- **Target Configuration Simplified** - Target configuration has been simplified to a name, type and then a series of key/value pairs for configuration. The complexity of the previous targets was unneccessary. If needed, we can extend the target configuration to support storing the original XML configuration as a part of the TargetConfig class, but we'll refrain from adding this, until it is needed.

- **Target Lifecycle Simplified** - Targets no longer are concerned with managing their own message queue. The previous version of NuLog maintained a queue both in the dispatcher, and in each target. Having queues in both places adde complexity, but didn't add any convenience or performance. The new version only has a queue in the dispatcher, targets are now "single threaded" in terms of lifecycle.
- **Observer Pattern for Configuration Removed** - The observer pattern for the previous system has been removed. Changes to the web configuration are handled in IIS by performing a blue/green recycle of the application, removing the need to "watch" the configuration for changes.

Getting Started (Quick Start)

If you are totally new to logging in software, first read *Introduction to Logging*.

If you are new to tag-based logging, read *Introduction to Tag-Based Logging*.

Step 1: Install Using NuGet

NuLog is delivered using NuGet. Look for, and install the NuLog package into any project you wish to use NuLog for. NuLog supports .Net versions 3.5, 4 and 4.5.2 (and on).

You'll want to get your configuration squared away. For more information on configuring NuLog, head over to *Configuration Template*.

Step 2: Get Standard Logger

```
// use NuLog
using NuLog;

namespace NuLogSnippets.Docs
{
    public class GetStandardLogger
    {
        // Get the logger from the log manager
        private static readonly ILogger _logger = LogManager.GetLogger();
        }
    }
```

Step 3: Log

```
using NuLog;

namespace NuLogSnippets.Docs
{
    public class GetStandardLogger
    {
        private static readonly ILogger _logger = LogManager.GetLogger();

        public void DoSomething()
        {
            // Log something
            _logger.Log("Hello, world!  I'll be dispatched later.");
            _logger.LogNow("Hello, world! I'll be dispatched immediately.");
        }
    }
}
```

Log vs. *LogNow*

NuLog is designed to defer the actual action of dispatching and logging a log event to a queue that is managed by a background process (thread). This allows control to return to the logging method much sooner, instead of having to wait for the log message to be dispatched before returning. When calling `Log` on the *Logger*, the log event is generated by the *Logger*, then added to a queue for dispatch by a background process. To dispatch a log event for immediate logging, use `LogNow` instead.

It is highly suggested that you use `Log` for almost everything, only using `LogNow` for log events that are more critically recorded.

The *Finalizer* on the class that dispatches the log events has been overridden to ensure that no new log events are queued, and any existing log events in the queue are dispatched, before the dispatcher is disposed. This means that there is a higher degree of certainty that queued log events are dispatched before the program exits. The only time queued log events won't be dispatched, is in case of a total crash of the application, or power failure - which would likely cause log event failure, anyways. If the nature of the log event is such that failure cannot be risked, just use `LogNow` for the event instead.

Configuration Template

Introduction

The standard way to configure NuLog is via the standard custom configuration section in your `web.config` or `app.config` file. By leveraging this configuration standard, NuLog benefits from the configuration management utilities

provided by .Net, including the ability to apply configuration transforms during build/deploy, as is popular with many *CI/CD* solutions.

Copy Pasta

Here's a bare-bones configuration for using NuLog. This will direct all log events to `Trace`:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
        <section name="nulog" type="NuLog.Configuration.ConfigurationSectionHandler, NuLog" />
    </configSections>
    <nulog>
        <targets>
            <target name="trace" type="NuLog.Targets.TraceTarget" />
        </targets>
        <rules>
            <rule
                include="*"
                targets="trace" />
        </rules>
        <tagGroups>
            <!--<group baseTag="base_tag" aliases="one_tag,two_tag,red_tag,blue_tag" />-->
        </tagGroups>
        <metaData>
            <!--<add key="meta" value="data" />-->
        </metaData>
    </nulog>
</configuration>
```

NuGet Package

Not available yet. There are plans to create a "development" NuGet package which will inject a sample NuLog configuration section into a `web.config` or `app.config`.

General Settings

Stack Frame

By default, the stack frame of the logging method is not included in log events. This stack frame is needed if you want to log the name of the method which made the log call (see *Layouts/Log Format* for more information).

To include the calling stack frame in the log events, turn it on on the main `<nulog>` element in the config:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
        <section name="nulog" type="NuLog.Configuration.ConfigurationSectionHandler,␣
↪NuLog" />
    </configSections>
    <nulog
        includeStackFrame="true"
        > <!-- ... -->
    </nulog>
</configuration>
```

Fallback Logging

By default, failures within NuLog are logged to *Trace*. To log to a text file, instead, set the *fallbackLog* to the file to log to. The path can be relative, or absolute:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
        <section name="nulog" type="NuLog.Configuration.ConfigurationSectionHandler,␣
↪NuLog" />
    </configSections>
    <nulog
        fallbackLog="nulog-fallback.log"
        > <!-- ... -->
    </nulog>
</configuration>
```

Rules for Routing

Introduction to Rules

Jumping straight in, here's an example rule configuration:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
        <section name="nulog" type="NuLog.Configuration.ConfigurationSectionHandler,␣
↪NuLog" />
    </configSections>
    <nulog>
        <!-- ... -->
        <rules>
            <rule
                include="one_tag,two_tag"
                exclude="red_tag,blue_tag"
                targets="trace,mytarget"
                strictInclude="false"
                final="false" />
        </rules>
        <!-- ... -->
    </nulog>
</configuration>
```

The `<rules>` section of the NuLog config can contain more than one `<rule>`. Rules are processed in order, from top to bottom. Let's walk through each of the attributes of a rule:

- **include** - A list of tags, comma separated, for include in the rule. One or all of these tags must be present in the log event for the rule to be considered a match.
- **exclude** - A list of tags, comma separated, for exclude in the rule. If any of these tags are present in the log event, the rule is considered not a match.
- **targets** - A list of targets, comma separated, that the log event is to be dispatched to if the rule is found to be a match. Targets are referenced by their configured name, not their type.
- **strictInclude** - If this flag is set, all of the tags listed in include must be present in the log event for the rule to be considered a match.
- **final** - A flag, that if set and this rule is considered a match, no further rules will be processed for the log event.

Wildcards

The `*` character can be used as a wildcard for matching tags in the *include* and *exclude* attributes. Wildcards match zero, or more, of any character. Wildcards can appear anywhere in the tag, and can be used more than once; for example: `*hello*world*`. In the *include* attribute, you can use a single wildcard: `*`, to match any (or all) tags.

Automatic Tags

There are some tags added automatically when logging:

- **Full Class Name** - The full class name, from which the log event was logged, is added as a tag to each call. This enables us to write rules that target any portion of the namespace, up to, and including the specific class name. For example, the tag `NuLog.CLI.PerfTune.Program` would be added when logging from a method in

the *Program* class in the *NuLog.CLI.PerfTune* namespace. One limitation here, is that tags are case insensitive, but C# is case sensitive. This means that if you have two classes in the same namespace that vary only by case, you won't be able to target just a single one, without adding extra tags to key off of.

- **Exceptions** - When logging an exception, the `exception` tag is automatically added to the log event, allowing us to target rules specifically to ones which contain exceptions.

Tag Groups

Introduction to Tag Groups

Tag groups were created to simplify managing tags and to allow for hierarchical tags. A list of tags are grouped under a single tag. Take a look at the configuration below as an example:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
        <section name="nulog" type="NuLog.Configuration.ConfigurationSectionHandler,␣
↪NuLog" />
    </configSections>
    <nulog>
        <!-- ... -->
        <tagGroups>
            <group
                baseTag="base_tag"
                aliases="one_tag,two_tag,red_tag,blue_tag" />
        </tagGroups>
        <!-- ... -->
    </nulog>
</configuration>
```

What this means:

- **baseTag** - The base tag that the alias tags equate to. If this tag is listed in a rule's `include` or `exclude` attributes, all of the tags listed as aliases would then qualify/satisfy the *include* or *exclude* requirement.
- **aliases** - The list of tags, comma delimited, which are considered the same as the named base tag.

An Impractical Example: Fruit

Perhaps you wanted to simply be able to say `fruit` in a rule to say that you wanted to send all log events containing *fruit* to be dispatched to a list of targets. You could create a tag group for the base tag `fruit`, and list under it `apple`, `pear`, `tomato`, and on.

Now, I know you aren't likely to be using fruit as tags, but this could be used to group all user actions together, for example: `useractions` could have the aliases `login`, `signout` and `resetpassword`. This becomes useful once you have more advanced tag routing, where you have multiple rules that would leverage the `useractions` base tag.

A Practical Example: Traditional Log Levels

Tag groups can be used to emulate the traditional behavior of log levels. This is done by equating all of the lower levels, to a given level:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
        <section name="nulog" type="NuLog.Configuration.ConfigurationSectionHandler,␣
↪NuLog" />
    </configSections>
    <nulog>
        <!-- ... -->
        <tagGroups>
            <group baseTag="trace" aliases="fatal,error,warn,info,debug" />
            <group baseTag="debug" aliases="fatal,error,warn,info" />
            <group baseTag="info"  aliases="fatal,error,warn" />
            <group baseTag="warn"  aliases="fatal,error" />
            <group baseTag="error" aliases="fatal" />
        </tagGroups>
        <!-- ... -->
    </nulog>
</configuration>
```

With these tag groups in place, we can then set a "level" to our rules:

```
<rule
    include="warn"
    targets="email" />
```

If we look at our `warn` tag group, it tells us that any log events with any of `warn`, `error` or `fatal`, will cause the rule to match, and be sent to the target named *email*.

Static Meta Data

Introduction to Static Meta Data

Static Meta Data is meta data that is added to every log event generated through the system.

```xml
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
        <section name="nulog" type="NuLog.Configuration.ConfigurationSectionHandler,␣
↪NuLog" />
    </configSections>
    <nulog>
        <!-- ... -->
        <metaData>
            <add key="Server" value="Web42B" />
            <add key="Env" value="Prod" />
            <add key="Release" value="1.2.42" />
        </metaData>
        <!-- ... -->
    </nulog>
</configuration>
```

Static Meta Data is useful for including information about the environment the logger is running on, helping to identify problems in your infrastructure. The benefit of including static meta data this way, is that these values can be transformed during a standard CI/CD process, being automatically updated by your favorite CI/CD tool, such as Octopus Deploy.

Updated Layout to Include Static Meta Data

Once you have your static meta data in place, you could include it in a custom layout for your targets:

```xml
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
        <section name="nulog" type="NuLog.Configuration.ConfigurationSectionHandler,␣
↪NuLog" />
    </configSections>
    <nulog>
        <!-- ... -->
        <targets>
            <target name="file" type="NuLog.Targets.TextFileTarget" path="app.log"
                    layout="${DateTime:'{0:MM/dd/yyyy hh:mm:ss.fff}'} | ${Release}-$
↪{Server}-${Env} | ${Tags} | ${Message}${?Exception:'\r\n{0}'}\r\n" />
        </targets>
        <!-- ... -->
    </nulog>
</configuration>
```

With the static meta data from the previous section, and this layout, `${Release}-${Server}-${Env}` would be rendered as `1.2.42-Web42B-Prod`.

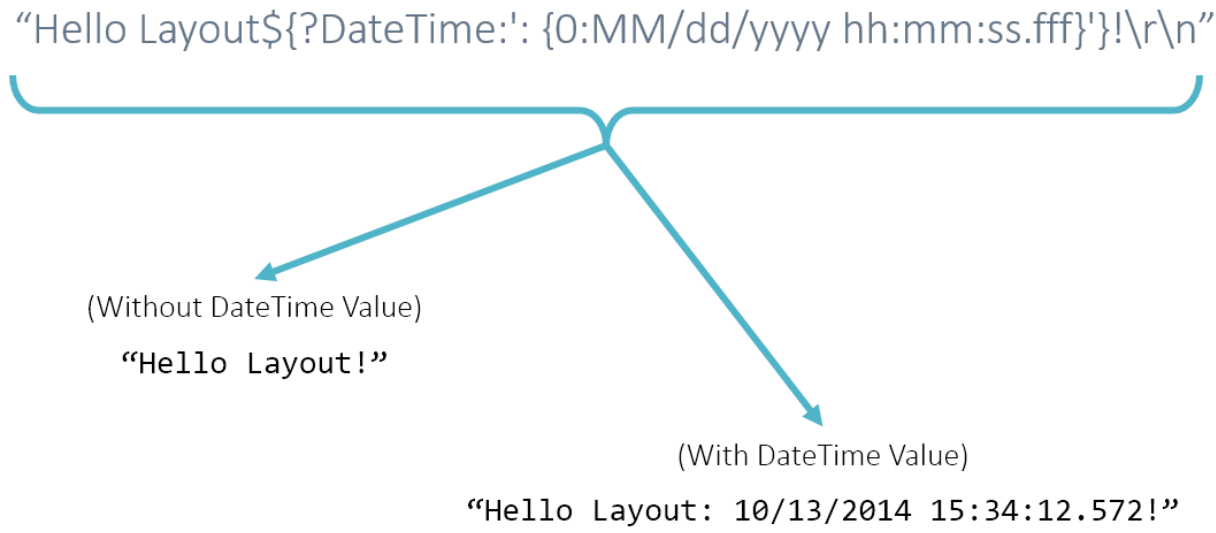
For more information about layouts, see *Layouts/Log Format* (up next).

Layouts/Log Format

Introduction to Layouts

Layouts are a mechanism for converting log events into text, and are used by many of the built-int targets, such as the *ConsoleTarget*, *TextFileTarget* and even the *MailTarget*. The default "standard" layout implementation uses a format string to define how the layouts are defined. Layouts are used by the standard text-based targets that are included with the framework (including the email target). Custom layouts can be implemented and used in the framework. This documentation focuses on the standard layout.

Layouts are a combination of static text and layout parameters. Static text is anything that is not wrapped in a property enclosure `${}`. Static text will always show in a log event formatted by the layout. Escaped characters are supported, and *suggested*, as newline characters are not automatically included at the end when a log event is being printed.



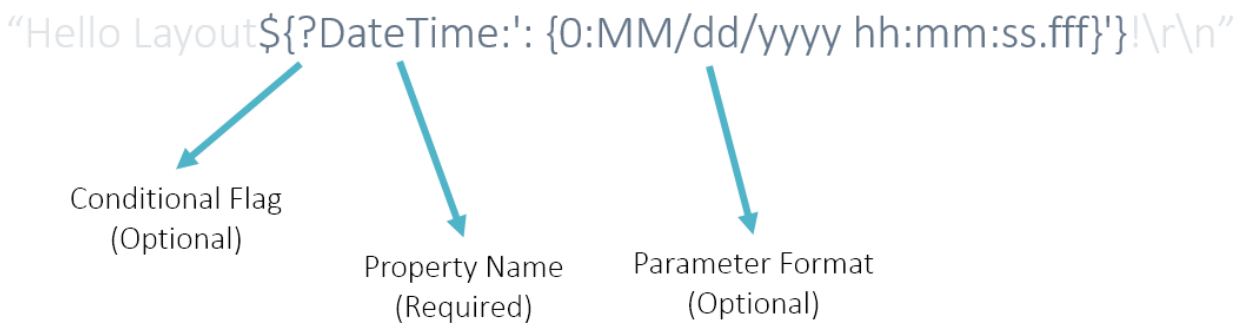
Special Parameters

There are some special parameters, that recieve special treatment when used:

- **${Tags}** - Produces a comma delimited list of all of the tags assigned to the log event.
- **${Exception}** - Produces special formatting of the exception associated with the log event - specifically, each inner exception will be listed, traversing down the chain.

Anatomy of a Layout Parameter

Parameters are wrapped with the property enclosure: `${}`. A Single parameter in the layout format refers to a single property in the log event. Parameters have three parts:

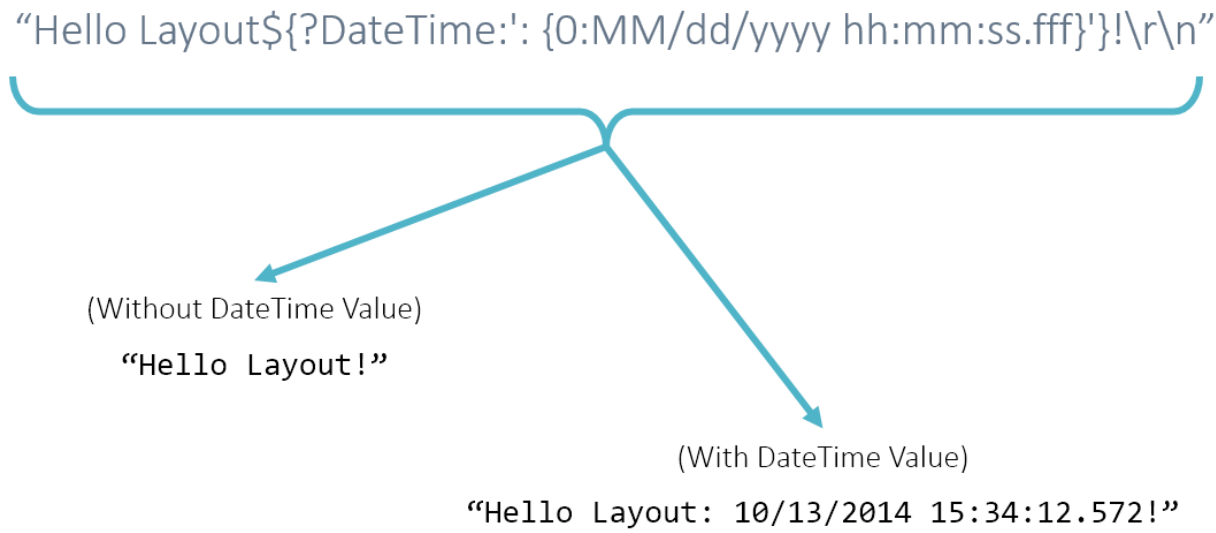


Let's explore:

1. **The Conditional Flag - *Optional*** - The conditional flag is a single `?` located at the front of the property, inside the property property enclosure `${}`. If the conditional flag is present, the property will only be included in the resulting text if the property is not null or empty.
2. **The Property Name - *Required*** - The name of the property within the log event is located at the beginning of the property string, after the conditional flag. The Property Name value is reflective and recursive, child values can be accessed using periods, for example: `DateTime.Day`. Any “Meta Data” associated with the log event is searched first for the property, by name. The log event is searched for the property by name, if the property is not found in the “Meta Data”.
3. **The Parameter Format - *Optional*** - The property format is used to format the value of the property which was evaluated from the log event. The property format is separated from the property name by a colon: `:`. The property format is wrapped in single quotes to allow for escaping within the format string. The framework uses `System.String.Format` with the property format and value.

Conditional Parameters

Using parameter formatting (which we will look at next), extra characters and text can be included in with a parameter. By using the conditional flag, you can tell the layout to only include the formatted text if the parameter identified by the name has a value. Take, for example:

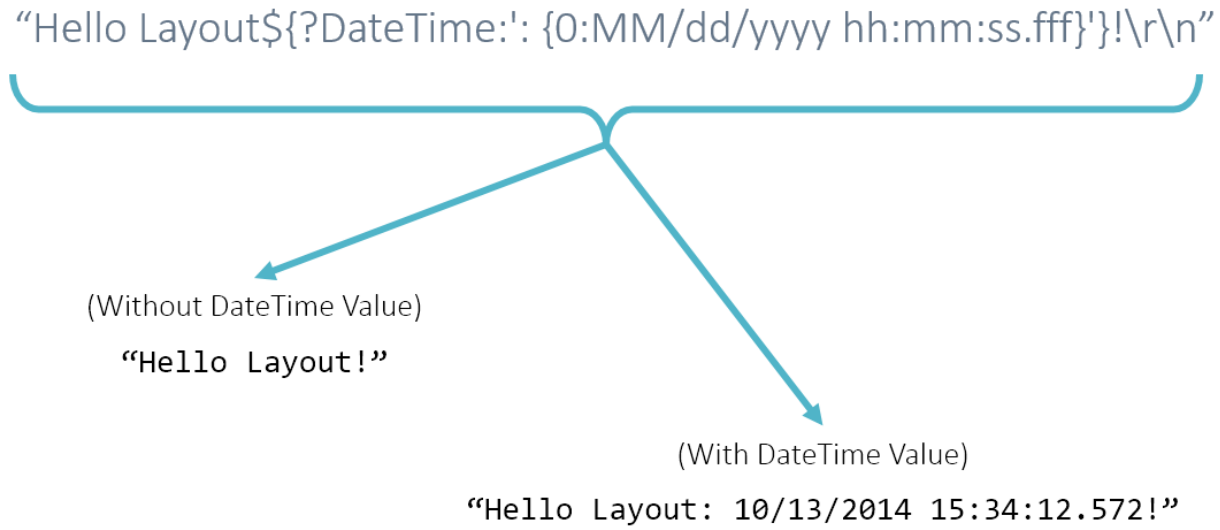


Notice how the resulting text could be different, between `Hello Layout!` and `Hello Layout: 10/13/2014 15:34:12.572!`. The two differences between these are the characters `:` and the date. Because of the

conditional flag `?` at the beginning of the parameter is set, the entire formatted text is not included when DateTime has no value, excluding the `:`. This prevents the formatted text from reading `Hello Layout:  !` or `Hello Layout:  null!` if no date is provided.

Formatted Parameters

The format portion of the parameter is separated from the parameter name by the `:` character. The format is wrapped in single quotes `'` and is passed as the pattern or format to the `System.String.Format` function. The first and only parameter passed for formatting is the matched parameter from the log event. The example below shows how we leverage the `System.String.Format` functionality for formatting a `DateTime`:



Also notice how in the format string, the date stamp is preceded by two characters - a colon and a space.

Targets

Configuration

NuLog comes with a number of built-in targets. Targets are configured within the `<targets>` section of the NuLog configuration:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
        <section name="nulog" type="NuLog.Configuration.ConfigurationSectionHandler,␣
↪NuLog" />
    </configSections>
    <nulog>
        <!-- ... -->
        <targets>
            <target name="mytarget" type="NuLog.Targets.TraceTarget" />
        </targets>
        <!-- ... -->
    </nulog>
</configuration>
```

All targets must have a `name` and a `type`:

- **name** - The *name* identifies the target to the various rules.
- **type** - The *type* indicates the concrete type of the target. When referencing the built-in NuLog targets, you can omit the assembly name from the type string; but if you are referencing a third party, or custom target, you'll need to include the assembly, as is standard when identifying a type: *"Some.Assembly.Namespace.Path.ToATarget, Some.Assembly"*.

Trace Target

NuLog.Targets.TraceTarget

The trace target writes log events to *Trace*:

```
<target name="mytrace" type="NuLog.Targets.TraceTarget"
        layout="${DateTime:'{0:MM/dd/yyyy}'} | ${Message}\r\n" />
```

The trace target has the following properties:

- **layout** - *Optional* - Defines the layout format for the target. By default, this is a standard layout, as documented in *Layouts/Log Format*.

Console Target

NuLog.Targets.ConsoleTarget

The console target writes log events to *Console*:

```
<target name="myconsole" type="NuLog.Targets.ColorConsoleTarget"
        layout="${DateTime:'{0:MM/dd/yyyy}'} | ${Message}\r\n"
        background="White"
        foreground="DarkBlue" />
```

The console target has the following properties:

- **layout** - *Optional* - Defines the layout format for the target. By default, this is a standard layout, as documented in *Layouts/Log Format*.

- **background** - *Optional* - An optional override to the Background Color of messages written to the console.
- **foreground** - *Optional* - An optional override to the Foreground Color of messages written to the console.

Text File Target

NuLog.Targets.TextFileTarget

The text file target writes log events to a text file:

```
<target name="mytarget" type="NuLog.Targets.TextFileTarget"
        path="app.log"
        layout="${DateTime:'{0:MM/dd/yyyy}'} | ${Message}\r\n" />
```

The text file target has the following properties:

- **path** - *Required* - The path to the text file to log to. Can be relative, or absolute.
- **layout** - *Optional* - Defines the layout format for the target. By default, this is a standard layout, as documented in *Layouts/Log Format*.

Mail Target

NuLog.Targets.MailTarget

The mail target sends log events via a SMTP server:

```
<target name="mytarget" type="NuLog.Targets.MailTarget"
        subject="Unhandled Exception in Super App!"
        to="me@superawesome.net"
        from="system@superawesome.net"
        smtpServer="mail.gtm.superawesome.net"
        body="The message proceeds: ${Message}"
        html="false"
        convertNewlineInHtml="false"
        smtpUserName="superuser"
        smtpPassword="awesomepass"
        enableSsl="true"
        smtpPort="993"
        smtpDeliveryMethod="Network"
        pickupDirectory="nope"
        timeout="1042" />
```

The mail target has the following properties:

- **subject** - *Required* - The subject line of the email. This is a layout format, as documented in *Layouts/Log Format*.
- **to** - *Required* - A semi-colon delimited list of email addresses to send the email to.
- **from** - *Required-ish* - Required if not set in the application/web config, in the `<system.net>` section. The email address to send the message *"from"*.
- **smtpServer** - *Required-ish* - Required if not set in the application/web config, in the `<system.net>` section. The network address of the SMTP server to send email messages through.

- **body** - *Optional* - An optional override for the layout format for the body of the email. By default, this is a standard layout, as documented in *Layouts/Log Format*.
- **html** - *Optional* - If set to *true*, signals that the body of the email is HTML. Defaults to *false*.
- **convertNewlineInHtml** - *Optional* - If set to *true*, replaces newline characters in the body of the email with `<br />` tags.
- **smtpUserName** - *Optional* - The username for authenticating to the SMTP server. If given, the password is required:
- **smtpPassword** - *Optional-ish* - The password for authenticating to the SMTP server. Required if the user name is given.
- **enableSsl** - *Optional* - If set to *true*, will use SSL when connecting to the SMTP server.
- **smtpPort** - *Optional* - The port to connect to the SMTP server on.
- **smtpDeliveryMethod** - *Optional* - The SMTP Delivery Method for sending the email through. Defaults to `Network`.
- **pickupDirectory** - *Optional-ish* - Required if *SpecifiedPickupDirectory* is set for the *smtpDeliveryMethod*. The path of the directory to which to write mail messages.
- **timeout** - *Optional* - The timeout value, in milliseconds, for when connecting to the SMTP server to send a mail message.

Advanced Usage

Shutting Down NuLog

Standard method

If for any reason you want to shut NuLog down before application exit, the most standard way to do this is using the *Shutdown* method of the *LogManager*. You would do this if you are using the *LogManager* to manage your logging construction needs:

```
1       public void DoStuff()
2       {
3           // ...
4
5           LogManager.Shutdown();
6
7           // ...
8       }
```

StandardLoggerFactory Finalizer

The standard logger factory implementation has a finalizer, which disposes the assets managed by the factory, namely, the *IDispatcher* assigned to it, when garbage collection deconstructs it.

Disposing the Dispatcher

It is important to dispose the dispatcher, so that the queued log events can be sent to their targets before application exit.

Like the *StandardLoggerFactory* the *StandardDispatcher* also has a finalizer, which will dispose the dispatcher when disposed of by the garbage collector.

Meta Data Providers

Much like static meta data, dynamic meta data can be included in log events to be delivered to the targets. To do this, a *IMetaDataProvider* instance must be passed to the log manager when requesting a logger.

Custom Meta Data Provider

The *IMetaDataProvider* interface has a single method for providing meta data:

```
    /// <summary>
    /// Defines the expected behavior of a meta data provider - namely, providing meta data.
    /// </summary>
    public interface IMetaDataProvider
    {
        /// <summary>
        /// Provide meta data for a log event.
        /// </summary>
        IDictionary<string, object> ProvideMetaData();
    }
```

There's not any more to it than that. As an illustration, however, let's take a look at an implementation that includes some request data for a MVC controller:

```
    public class MyControllerMetaDataProvider : IMetaDataProvider
    {
        private readonly Controller myController;

        public MyControllerMetaDataProvider(Controller controller)
        {
            myController = controller;
        }

        public IDictionary<string, object> ProvideMetaData()
        {
            var request = myController.Request;
            return new Dictionary<string, object>
            {
                { "UserHostAddress", request.UserHostAddress },
                { "URL", request.Url }
            };
```

```
        }
    }
```

This meta data provider will include information specific to the request, for the given controller. Next, let's take a look at how to use our new meta data provider.

Using Meta Data Providers

To use our custom meta data provider, we pass an instance of it to our log manager when requesting a logger:

```
    public class MyMetaDataProviderController : Controller
    {
        private readonly ILogger myLogger;

        public MyMetaDataProviderController()
        {
            var myMetaDataProvider = new MyControllerMetaDataProvider(this);
            myLogger = LogManager.GetLogger(myMetaDataProvider);
        }

        public ActionResult Index()
        {
            myLogger.Log("Will include my custom, request specific meta data.", "index
↪");
            return View();
        }
    }
```

Advanced Rule Routing

The 'final' and 'strictInclude' Flags

Complex routing can be achieved using the *final* and *strictInclude* flags:

- **final** - When set to *true*, and if a log message matches the rule, no further rules will be processed. Rules are processed in consecutive order, as listed in the rule list.
- **strictInclude** - By default, if any tag is matched in the *include* list, the rule is matched. By seting the *StrictInclude* flag, all tags (or tag patterns) listed in *include* must match for the rule to be considered a match.

Examples

final

Let's say that you want to send messages with the "consoleonly" tag, only to console, but have a general, catch-all rule in place. In the following example, events with the *consoleonly* tag will be routed only to the console target, all others will only go to the file target:

```
  <rules>
    <rule
      include="consoleonly"
      targets="console"
      final="true" />
    <rule
```

```
      include="*"
      targets="file" />
  </rules>
```

strictInclude

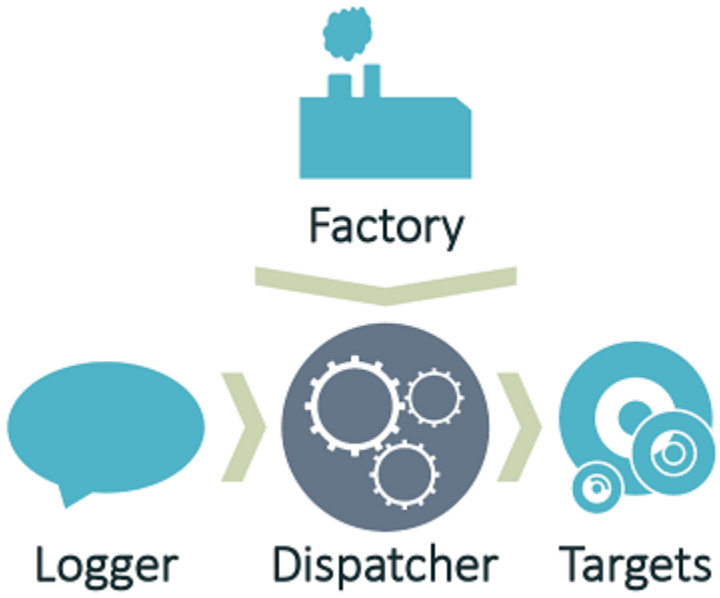
We use the *strictInclude* flag to state that all tags/patterns in the "include" of the rule must be matched for the rule to match. This enables us to zero in on specific log events:

```
  <rules>
    <rule
      include="NuLogSnippets.*,index"
      strictInclude="true"
      targets="file" />
  </rules>
```

The above rule states that all log events with the *index* tag, and a tag starting with *NuLogSnippets.* are routed to the target named *file*. Note that the class that log events are logged from are automatically added as tags, for example: *NuLogSnippets.Docs.MyMetaDataProviderController* is added to the log events generated by the *Index* action, and any other calls to the logger, in that class.

Architectural Overview

Factory, Logger, Dispatcher and Targets



Overview

- **Factory** - Responsible for building loggers, the dispatcher, and everything the dispatcher depends on. NuLog conforms to the 'Dependency Inversion Principle <https://en.wikipedia.org/wiki/Dependency_inversion_principle>'_, forcing all the construction up to the factory.
- **Logger** - Constructed by the factory, and given a reference to a dispatcher, the logger is responsible for building log events and passing them to the dispatcher.
- **Dispatcher** - Responsible for receiving log events and managing them in a queue, telling them to log to certain targets at the appropriate times, based on the configured rules.
- **Targets** - Targets are responsible for writing log events to their medium, such as trace, console, text file, or a plethora of others.

Log Event Lifecycle

1. Application code calls a *Logger*, sending data over to be converted into a *LogEvent*.
2. The *Logger* converts the data into a *LogEvent* and passes it to the logger's *Dispatcher*, either for immediate, or deferred dispatch.
3. The *Dispatcher* receives the *LogEvent*, and either sends it immediately, or places it into a queue to be dispatched later.
4. The *Dispatcher*, either immediately, or as it works through its queue, figures out which targets each log event need to go to, based on the tags on the log event, and the configured rules.

5. The *Dispatcher* then tells the *LogEvent* to present itself to the appropriate *Target* instances for writing.

Architectural Principles

Here are our guiding principles, for the NuLog project. These principles come together to form the "philosophy" of the project:

1. **Honest Mistakes, and Honesty in General** - We tolerate people's honest mistakes. In fact, we encourage them. This means malicious finger-pointing is not tolerated, and may get you banned from the project. It is up to me (Ivan Pointer), to decide if someone is being malicious, or not. I want people to feel the freedom needed to take risks, honestly. I expect high integrity of all our official members.
2. **Simple** - Period. Avoid introducing complexity until it is *necessary*, and no sooner. It is the simplicity of the framework that gives it power.
3. **NuLog is a Logging Framework** - Not a message bus or queue. NuLog shouldn't be handling notifications out to your end users, or acting as a bus to deliver messages between your application's tiers.
4. **Unassuming** - NuLog needs to stay out of the way. If NuLog breaks, it needs to break "silently" without letting exceptions bubble up. Leveraging NuLog cannot get in the way of the developer, but must instead, support the developer. NuLog is a useful tool. NuLog must not be a helpful tool. Useful tools require the developer to take action, whereas helpful tools take action on behalf of the developer.
5. **Tag-Based** - Offers more simplicity, and more flexibility, than the traditional "level" based logging.
6. **Extensible** - Build using the 'Dependency Inversion Principle <https://en.wikipedia.org/wiki/Dependency_inversion_principle>'_. Developers who leverage the framework must be able to replace any portion of the framework with their own pieces, given that they conform to the 'Liskov Substitution Principle <https://en.wikipedia.org/wiki/Liskov_substitution_principle>'_.
7. **SOLID Principles** - We adhere to the 'SOLID Principles <https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)>'_, as best we reasonably can.
8. **Business-Driven-Development** - We believe that the best documentation of a system are well written unit and integration tests. We demand that there be *near* complete code coverage, and the checks must be high quality. All code that can be tested, must be done so before said code is written. Tests are first to document the expected behavior of the code, and second, have an ansilary benifit of verifying that the implementation actually does what is expected. * **Any code (pull requests, etc.) that is not properly covered will be rejected.** *
9. **Continuous-Integration** - Continuous integration is a **must**. We recognize and accept that CI is a discipline, and not a set of tools. We embrace completely the discipline of CI. **This means that if you break it, you fix it:** If you break the build, you are expected to fix the build. We don't tolerate a broken build being left to rot: if the build breaks, all other changes/work stops until the build is fixed.
10. **Perfromant** - Performance is a high priority of the NuLog project. This said - code first for readability and maintainability. After this, use performance analysis tools to identify hot-spots in the code, and very clearly document any "smells" that are *necessary* for performance. If it isn't *necessary*, don't do it.
11. **No External Dependencies** - No third-party libraries should be required to use the core of NuLog. A logging framework shouldn't pull in any other dependencies, and should be unassuming. This means no *IoC* containers, or even a JSON parsing library. The core project needs to only reference the standard .Net assemblies.
12. **External Dependencies in Extension Libraries** - Extenral dependencies need to be brought in through extension libraries. An example of this would be a separate project for a target which posts log events to a Slack channel.
13. **NO TOLERANCE FOR GPL. PERIOD.** - GPL is not to come within 1,000 miles of NuLog. GPL is a cancer the likes this world hasn't seen before, and as *mamma says, GPL is the devil*. But-**NO. PERIOD.** We prefer MIT and Apache 2.0 around here. This same policy applies for any *copyleft* style licenses, which are designed to restrict your freedom and rights, as opposed to protecting them.

Architectural Policies

These policies are more fluid than the principles, and are therefore, more flexible, and change more frequently. They are also more detailed in nature. These help to refine the vision set forth by the principles:

1. **Fallback Loggers Should be Independent** - The fallback loggers should not leverage the targets, or other parts of the NuLog system, to perform their duties. The functionality of the fallback loggers need to be completely contained within the fallback loggers. The reason for this is simple: we need the fallback loggers to work, even when targets and other things are breaking. If the fallback logger depends on a target's implementation, when that target fails, so will the fallback logger, and consequently, the developer will not be informed of the failure.
2. **Using FakeItEasy for a mocking framework** - I was tempted to not even use a mocking framework, especially once I saw that Moq had a BSD license. After a little searching, I found FakeItEasy, which is under the MIT license, and has had a fairly active community. Adding a mocking framework won't add any dependencies to NuLog itself, as the tests aren't distributed with the library. FakeItEasy will definitely decrease complexity. Between the loose coupling, the reduction in complexity, and the friendly license, I've decided to leverage FakeItEasy for some of the more complex tests.
3. **Text File Target: No rotation/archiving** - There are a lot of different log aggregation frameworks that handle this, and, it's pretty easy to set this up with PowerShell, Batch, Shell, etc. This is generally something that system administrators prefer to manage externally. Because of this, the text file target's focus will be to be very "hands off" of the text file it produces - avoiding keeping file handles open, etc. - to play nicer with a separate process performing log management.
4. **Simple Email Target: This is for logging, not message queuing** - Not only does it significantly increase the complexity of the target, but for a purpose I believe to be out of scope of the core purpose of NuLog. Adding "advanced" features would encourage the abuse of the logging system, as use as a notification engine - which NuLog is not. This doesn't prevent a later "extended" email target (as an add-on package, or 3rd party contribution, perhaps).

Custom Targets

We believe that the most common customization of NuLog that will be sought after, will be custom targets. Here's what it takes.

Where to Start?

Aiming to be as flexible as possible, there are many different layers of abstraction to our targets. This allows you to create targets with just the amount of functionality you desire.

As an example, let's take a look at the inheritance tree of our *ConsoleTarget*:

ConsoleTarget > *LayoutTargetBase* > *TargetBase*, *ILayoutTarget* > *ITarget*

- **ConsoleTarget** - The console target implementation.
- **LayoutTargetBase** - Provides standard layout parsing and instantiation functionality, for leverage by implementing targets in their *Write* methods.
- **ILayoutTarget** - Defines the base expected behavior of a target with a layout.
- **TargetBase** - Provides an empty implementation of the *IDisposable* pattern, and a couple helper methods for configuration.
- **ITarget** - Defines the expected behavior of a target.

Bare Minimum: Implement the ITarget Interface

The ITarget Interface

A target must implement the *ITarget* interface. This states that a target must:

- Be disposable (*IDisposable* interface)
- Have a Name
- Must be able to recieve configuration.
- Must be able to recieve a log event for writing.

```
public interface ITarget : IDisposable
{
    /// <summary>
    /// The name of this target, which is used to identify this target in the various rules.
    /// </summary>
    string Name { get; set; }

    /// <summary>
    /// Tells the target to configure itself with the given config.
    /// </summary>
    void Configure(TargetConfig config);

    /// <summary>
    /// Write the given log event.
    /// </summary>
    void Write(LogEvent logEvent);
}
```

The lowest level implementation of a target may look something like this:

```
public class HelloWorldTarget : ITarget
{
    public string Name { get; set; }

    public void Configure(TargetConfig config)
    {
        // Nothing to do
    }

```

```
    public void Write(LogEvent logEvent)
    {
        Debug.WriteLine(logEvent.Message);
    }

    #region IDisposable Support

    protected virtual void Dispose(bool disposing)
    {
        // Nothing to do
    }

    // This code added to correctly implement the disposable pattern.
    public void Dispose()
    {
        Dispose(true);
    }

    #endregion IDisposable Support
}
```

Using Your Custom Target

Once you've created your custom target, you need to reference it in your config:

```
<nulog>
  <!-- ... -->
  <targets>
    <target name="myCustomTarget"
            type="NuLogSnippets:NuLogSnippets.Docs.CustomTargets.HelloWorldTarget" />
  </targets>
  <!-- ... -->
</nulog>
```

Less is More: Extend the TargetBase

You can actually implement your custom target in fewer lines of code by leverage the given *TargetBase* abstract class:

```
public class HelloWorldShortTarget : TargetBase
{
    public override void Write(LogEvent logEvent)
    {
        Debug.WriteLine(logEvent.Message);
    }
}
```

Most Useful: Extend the LayoutTargetBase

Since it is text data that we're generally logging, the abstract class that provides the best starting point for most custom targets, will be the *LayoutTargetBase*. With the *LayoutTargetBase*, you get:

- *Everything you get with the 'TargetBase', plus:*
- A *protected ILayout* instance that is automatically configured using the *layout* attribute from the config.

The *StandardFactory* specifically recognizes the *ILayoutTarget* interface, which has a special *Configure* method which recieves an instance of *ILayoutFactory*. This inverts the dependency on *ILayoutFactory*, allowing it to be injected into the target from above, in compliance of the *Dependency Inversion Principle*.

When extending the *LayoutTargetBase*, you get access to the automatically configured *layout* member:

```
public class HelloLayoutTarget : LayoutTargetBase
{
    public override void Write(LogEvent logEvent)
    {
        var formattedText = this.Layout.Format(logEvent);
        Debug.Write(formattedText);
    }
}
```

Now, when referencing your custom target in your config, you can set the layout:

```
<target name="myCustomTarget"
           type="NuLogSnippets:NuLogSnippets.Docs.CustomTargets.HelloLayoutTarget"
           layout="${DateTime:'{0:MM/dd/yyyy}'} | ${Message}\r\n" />
```

Target configuration

If you're building a custom target, chances are, you'll need some kind of custom configuration for your target.

Configuration Interface

The *ITarget* interface defines a method for receiving configuration from the factory:

```
void Configure(TargetConfig config);
```

This method recieves a *TargetConfig* object, which contains the *Name* of the target, the *Type* name of the object, and a *Dictionary* of *Properties*.

The properties are read from the XML configuration, and are the attributes of the XML *target* element from the config. Consider the following example:

```
<target name="mytarget"
           type="NuLogSnippets:NuLogSnippets.Docs.CustomTargets.CustomConfigTarget"
           myCustomProperty="Yellow, World!"/>
```

The *myCustomProperty* property would be included in the *TargetConfig* object's *Properties Dictionary* with a key of "*myCustomProperty*" and a value of "*Yellow, World!*".

myCustomProperty could then be retrieved in our configure method, like so:

```
public void Configure(TargetConfig config)
{
    this.MyCustomProperty = Convert.ToString(config.Properties["myCustomProperty"]);
}
```

Configuration Helpers in TargetBase

The *TargetBase* abstract class provides a couple helpers to make accessing properties a bit easier:

```
public override void Configure(TargetConfig config)
{
    base.Configure(config);

    this.MyStringProperty = GetProperty<string>(config, "oneFish");

    bool isTwoFish;
    this.MyBoolProperty = TryGetProperty<bool>(config, "twoFish", out isTwoFish) &&␣
↪isTwoFish;
}
```

Logger Factories and the Log Manager

Introduction

The *Log Manager* is the entry point for the NuLog framework - it is where developers and their applications most commonly request loggers.

The *Log Manager* contains a reference to an instance of *ILoggerFactory*, the *StandardLoggerFactory* by default. The *Log Manager* leverages the *ILoggerFactory* to build the loggers for the developers and their applications.

Developers don't have to use the *Log Manager* to build their NuLog relics, instead going directly to a *ILoggerFactory*, or even building the loggers themselves (not suggested). A better approach, would be to extend the *StandardLoggerFactory* to provide your own implementations of the StandardLoggerFactory components that make up the typical NuLog architecture.

ILoggerFactory Interface

First, let's take a look at the *ILoggerFactory* interface, which defines the expected behavior of a logger factory.

```
/// <summary>
/// Defines the expected behavior of a logger factory. The logger factory is␣
↪responsible for
/// providing instances of the various parts of the NuLog system.
/// </summary>
public interface ILoggerFactory : IDisposable
{
    /// <summary>
    /// Gets a logger.
    /// </summary>
    ILogger GetLogger(IMetaDataProvider metaDataProvider, IEnumerable<string>␣
↪defaultTags);
}
```

That's it! The sole job of a logger factory is to construct instances of loggers. Done.

Duties of the Log Manager

The log manager has three primary duties:

- **Retrieval of Loggers** - The *LogManager* is the main point of entry for developers and their applications to retrieve loggers.
- **Overriding the Default ILoggerFactory** - The *LogManager* provides a way to inject in your own custom implementation of *ILoggerFactory*, to enable you to customize any piece of the framework.
- **Shutdown NuLog** - The *LogManager* provides a hook for disposing the assigned *ILoggerFactory*, which will shutdown the NuLog system, flushing all queued log events, and not allowing any more to enter the queue.

Overriding the Default Logger Factory

The most conventional way to customize NuLog, is to create your own implemenmtations of the various pieces of NuLog, and extend the *StandardLoggerFactory*, finally assigning it as the factory for the *LogManager*:

```
1  public class SetLogManagerLoggerFactory
2  {
3      public void SetupMyApplication()
4      {
5          LogManager.SetFactory(new MyCustomLoggerFactory());
6      }
7  }
```

Extending the Standard Logger Factory

Introduction

The "standard" logger factory is named such, because through the log mnanager, and a custom logger factory, you can change almost any part of NuLog. I suggest, however, only changing those parts which you need to change, the rest of NuLog is very carefully thought and thuroughly tested.

To customize the behavior of NuLog, you would create your own implementation of certain parts of the NuLog system, extend the *StandardLoggerFactory* to leverage your custom implementation, then set your extension of the logger factory into the *LogManager*, as is discussed in *Logger Factories and the Log Manager*.

The first thing to consider is that the *StandardLoggerFactory* has a constructor that receives the *Config* of the system. This means that the *Config* is available to each of the members.

I suggest using as much of the standard implementation as possible, as performance and stability have been carefully built into them. While you can completely customize NuLog, it would be prudent to take advantage of the work we've done here in the standard implementation.

Standard implementation

MakeDispatcher()

The *MakeDispatcher* method is responsible for creating the dispatcher for the system. Overload this to implement your own dispatcher implementation.

MakeTargets()

The *MakeTargets* method is responsible for creating the various targets based on the config, returning a collection of targets.

MakeTagGroupProcessor()

Tag group processors are responsible for determining aliases between tags, such as *apple* and *tomato* being synonymous for *fruit*.

MakeRuleProcessor()

Rule processors are responsible for determining which targets a set of tags correspond to, based on the parsed rules from the config. The rule processor is used by the tag router.

MakeTagRouter()

Responsible for indicating which targets a set of tags go to, unlike the rule processor, the tag router has no knowledge of rules, but instead will have some caching and performance considerations built in. In the standard implementation of NuLog, a rule processor is given to the tag router, and the dispatcher works directly with the tag router, having no knowledge of the rule processor.

MakeTagNormalizer()

The tag normalizer is responsible for normalizing the tags, such as making them lower-case and trimming white space, and converting special characters.

MakeLayoutParser()

Layout parsers are responsible for taking a string representation of a layout, and converting it into a concrete list of layout parameters.

MakePropertyParser()

Property parsers are responsible for reflectively traversing a given object, and returning the value of an identified property, based on the list of property names given.

MakeLayout()

Layouts are responsible for converting a log event to a string representation. The standard layout uses the concept of layout parameters to achieve this. If you're implementing your own layout, you may need your own layout parser, and my not even need the concept of layout parameters.

MakeFallbackLogger()

The fallback logger is used when a failure occurs during the NuLog lifecycle. It is supposed to fail silently, as it is the fallback for other failures in the system. Without a properly behaving fallback logger, NuLog may leak failures into the implementing code.

Example Extension

Introduction to Logging

The Basics

- Logging consists of two simple elements: log events and log targets. For example, a family's signature on a guest role at a wedding, or a security guard's notes in his notebook.
- In software, logging is primarily used for troubleshooting/debugging and auditing purposes.
- A logging framework is responsible for providing a structured way to get log events into log targets.

Log Events

- Log events are messages or objects to be delivered to a log target (the family's signature or the security guard's note)
- Log events contain information about an event that happened
- A log event can be a simple message like "Hello, World!", or a complex multilevel object that represents a web request

Log Targets

- Log targets receive or store log events
- A log target can display a log event immediately, such as in a terminal window, or store the log event to persistent storage, such as in a file or database
- Log targets can also interface web services or email clients, delivering the log event to an external destination
- A single log target is generally responsible for a single destination. For example, a single "Text File Target" would write to a single file
- A single application will likely have multiple targets

Introduction to Tag-Based Logging

Traditional logging frameworks log based on a "level" (in terms of loudness, or volume), such as *trace*, *debug*, *info* and up to *error* and *fatal*. A sensitivity is configured, and anything equal to, or louder than the sensitivity, would be logged. This style of logging allows for "filtering" the log messages based on the amount of noise desired. This kind of filtering doesn't naturally support being more selective in which messages are filtered.

Tag-based logging facilitates a far-higher level of control in filtering log messages. The various "levels" can be represented by tags, but also, so much more. Instead of being limited to "loudness", log events can be tagged with any trait needed, such as the type of request being processed, or even the roles of the user associated with the event. The possibilities are much more free than with traditional logging frameworks.

NuLog follows a basic process for handling log events:

- **Tags** are assigned to log events
- **Rules** determine which log targets specific log events are dispatched to using the assigned tags. Not all log events are dispatched to all log targets
- **Targets** are responsible for handling the log events dispatched to them

---

Tags

Tags can represent practically anything pertaining to a log event:

- A particular target, such as "database" or "file"
- A particular status or event, such as "exception" or "authenticated"
- A particular source of log events, such as "SomeMVCApp.SomeController"
- NuLog automatically includes the full class name of the logging class as a tag on the log event

- Any other helpful grouping!

Targets

A single target represents a single destination for log events. A target can represent:

- A text file
- A database table
- A web service for logging
- A console window
- A trace log
- Any other custom log event destination

Rules

Rules define which targets a log event is dispatched to using the tags associated to said log event. Rules can be defined with:

- Which tags are included in the rule (include)
- Which tags are excluded from the rule, (specifically from those selected by "include") (exclude)
- Whether or not all tags defined in "include" must be present in the log event for the log event to match the rule (strictInclude)
- Which targets matching log events are to be dispatched to (targets)
- Whether or not to process any further rules for the log event if the current rule matches (final)

About NuLog and the Team

About NuLog

Working on some larger systems for state government, my friend Myke and I....

Development Team

Right now, just Ivan Pointer, but would love to have some more join to contribute maintain...

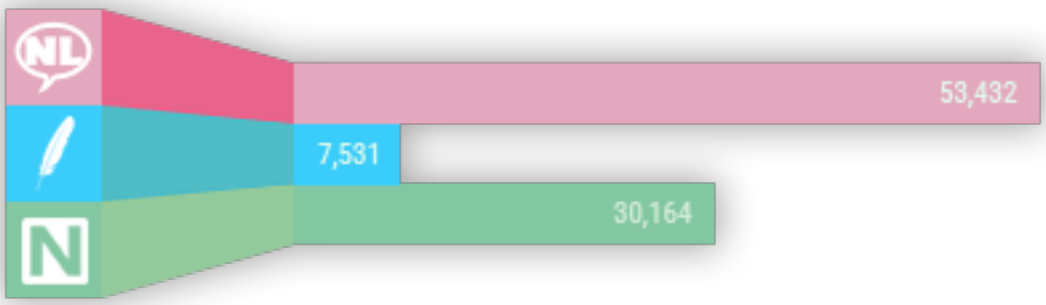
Performance

The initial build of NuLog focuses on stability and usability. Future releases will focus on benchmarking and performance more.

This said, NuLog still compares well to the other leading logging frameworks.

Benchmarking

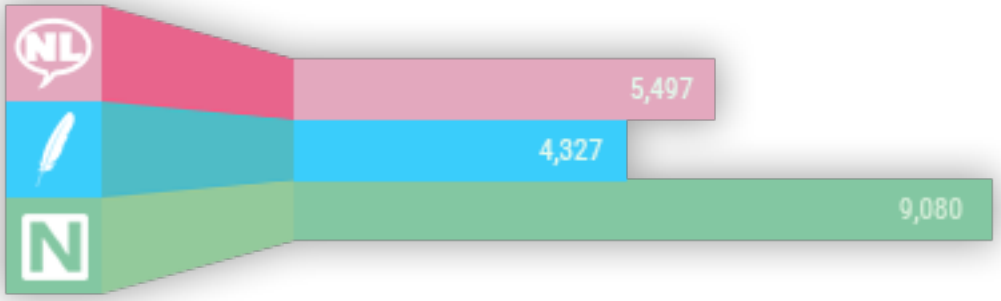
Trace Benchmark



Writing 10,000 log events to trace, average of 10 samples:

- **NuLog** - Writes at a rate of 53,432 messages per second.
- **Log4Net** - Writes at a rate of 7,531 messages per second.
- **NLog** - Writes at a rate of 30,134 messages per second.

Console Benchmark



Writing 10,000 log events to console, average of 10 samples:

- **NuLog** - Writes at a rate of 5,497 messages per second.
- **Log4Net** - Writes at a rate of 4,327 messages per second.
- **NLog** - Writes at a rate of 9,080 messages per second.

Benchmark Interpretation

The following assumptions are made about the trace and console benchmarks, taken together:

- NuLog has a very low overhead in routing log events to targets, when compared to the other logging frameworks.
- NuLog has room to improve in performance tuning for individual targets.
- NuLog falls between Log4Net and NLog in this simplistic performance comparison.

License (MIT)

The MIT License (MIT)

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.