

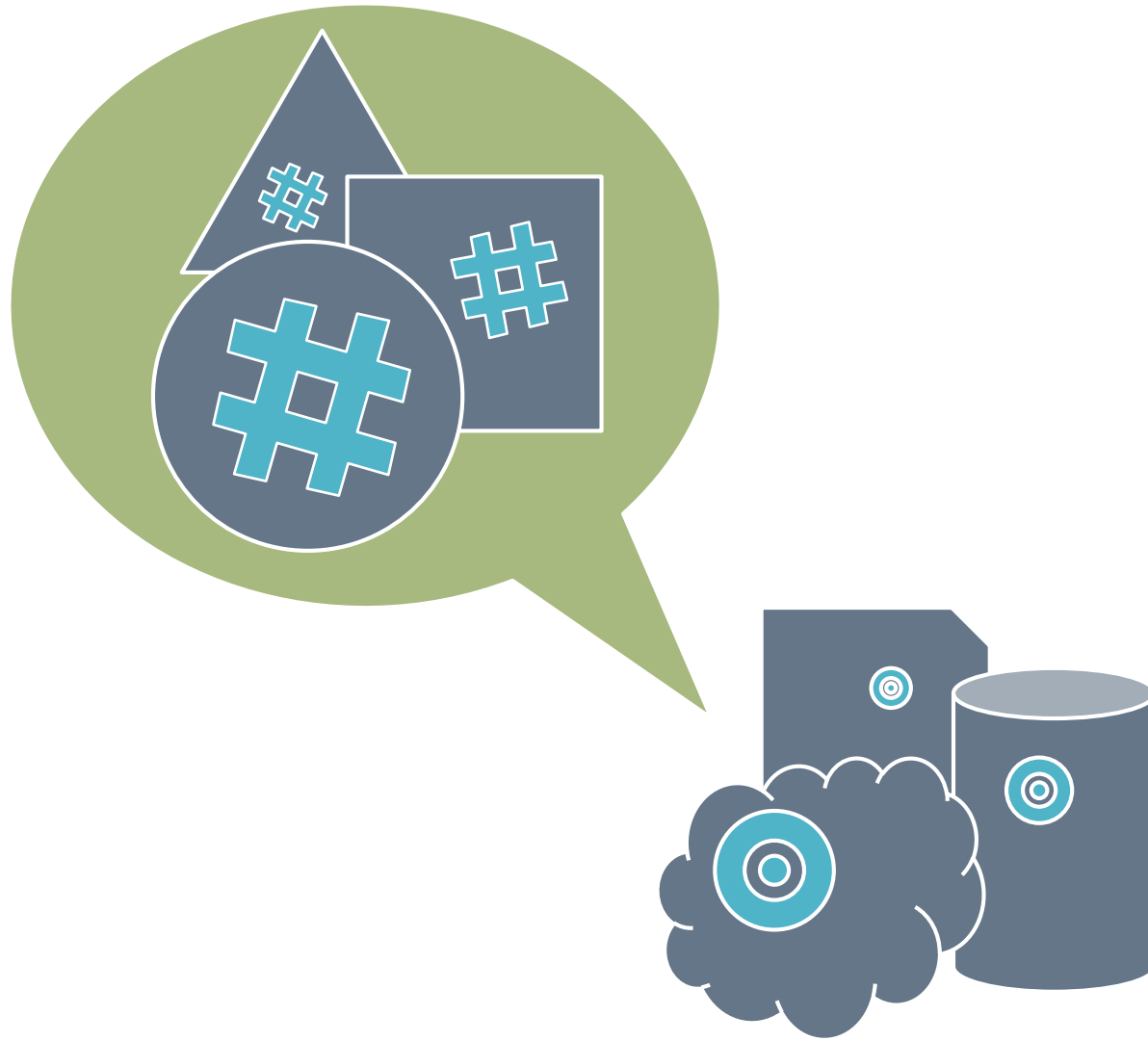
NuLog

Like all magnificent things, it's very simple



Log Events and Log Targets

Simplicity is the ultimate sophistication

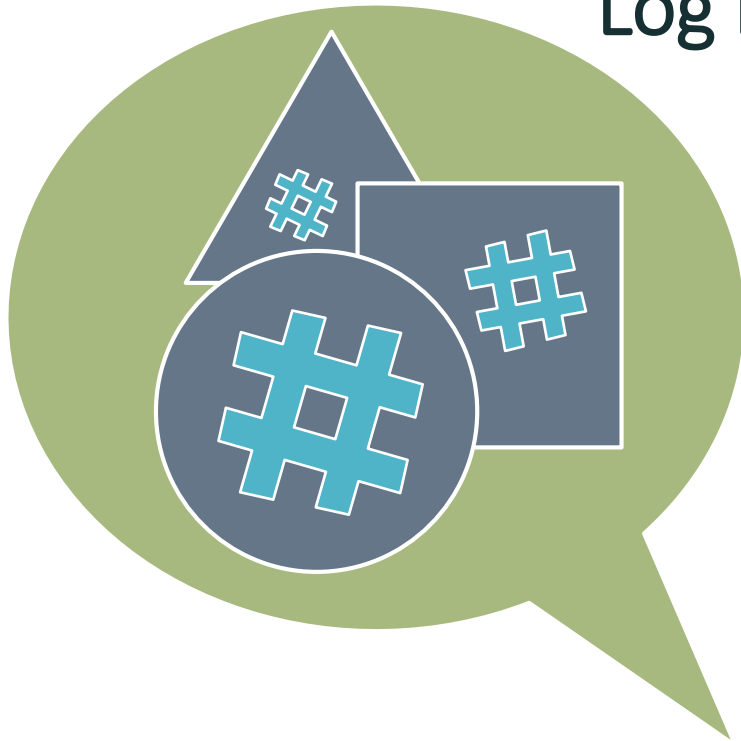


What is NuLog?

NuLog is a:

- Simple
- Effective
- Light-weight
- Extensible
- Tag-based logging framework
- *(No, it won't make your coffee for you)*

Log Events



Log Targets



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
- 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

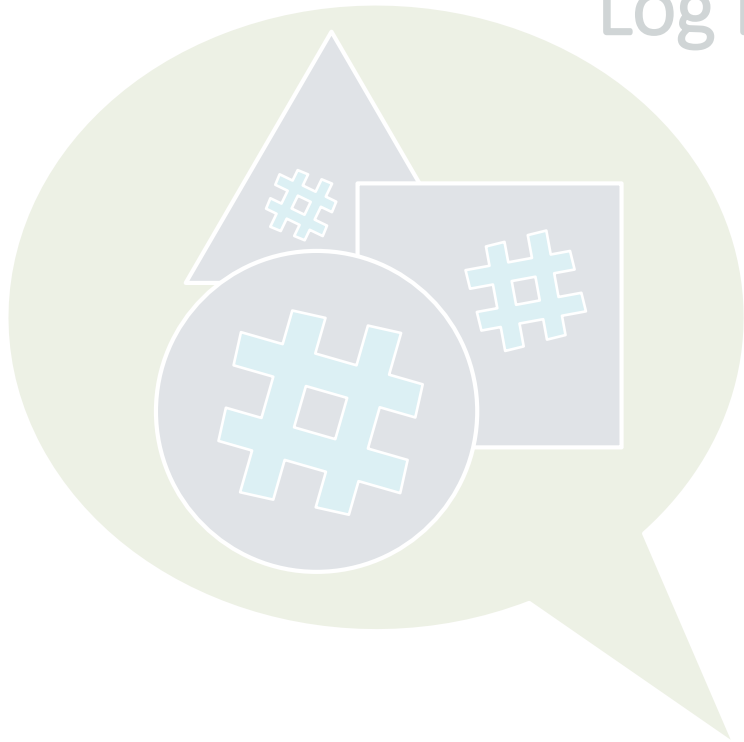
Log Events



Log Events

- Log events are messages or objects to be delivered to a log target
- Log events contain information about an event that is to be displayed to a user, or saved to a destination
- A log event can be a simple message like “Hello, World!” or a complex multi-level object that represents a HTTP POST request

Log Events

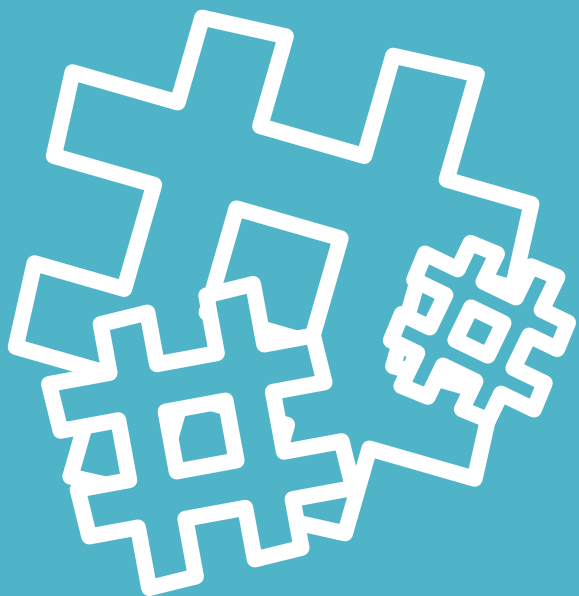


Log Targets

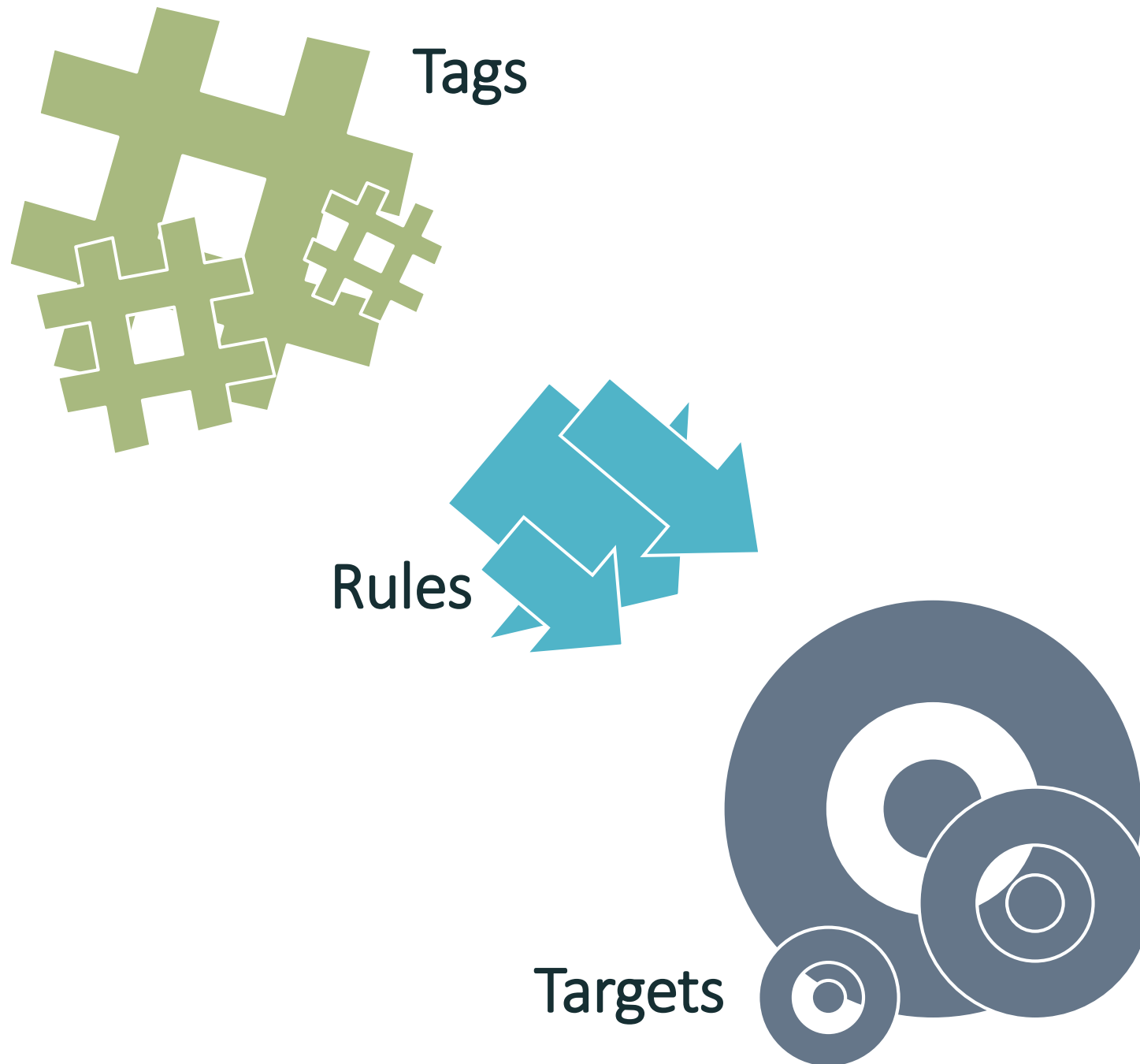


Log Targets

- Log targets receive 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 a third party destination
- A single log target is generally responsible for a single destination; For example, a single “File Target” would write to a single file

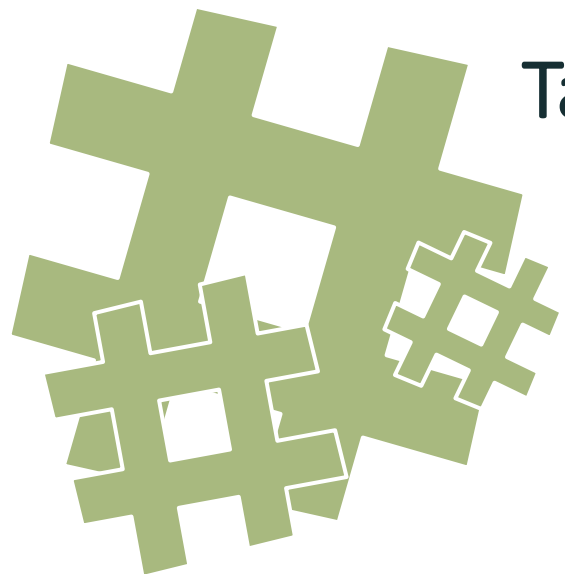


Tag-Based Logging
Simplicity is the ultimate sophistication



Tag-Based Logging

- There can be multiple log targets in a software system, and therefore: we need a way to define which log events go to which log targets. NuLog uses a tag-based approach:
 - 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

Rules



Targets



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/messages, such as¹:
`SomeMVCApp.SomeController`
 - Any other helpful grouping!

1. NuLog automatically includes the full class name of the calling object as a tag on the log event.



Tags

Rules



Targets



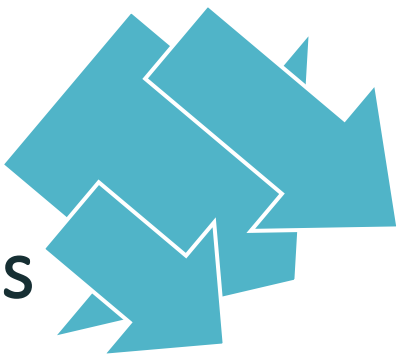
Targets

- A single target represents a single destination for log events
- A target can represent:
 - A rolling text file
 - A database table
 - A web service for logging
 - A console application
 - A Trace Log
 - Any other custom log event destination



Tags

Rules

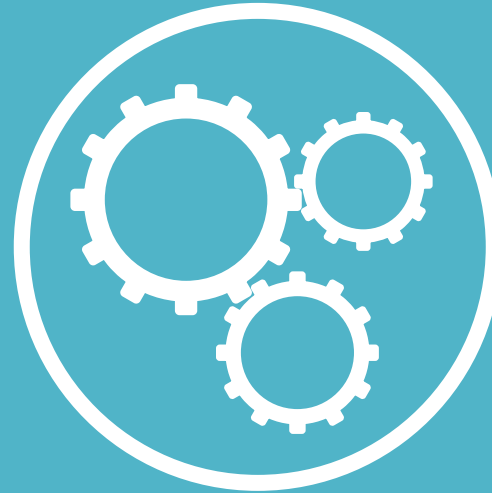
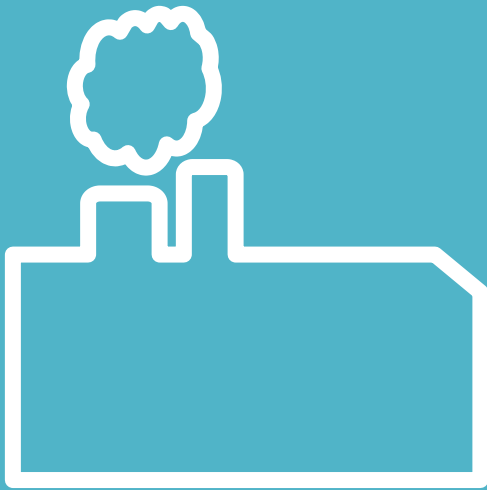


Targets



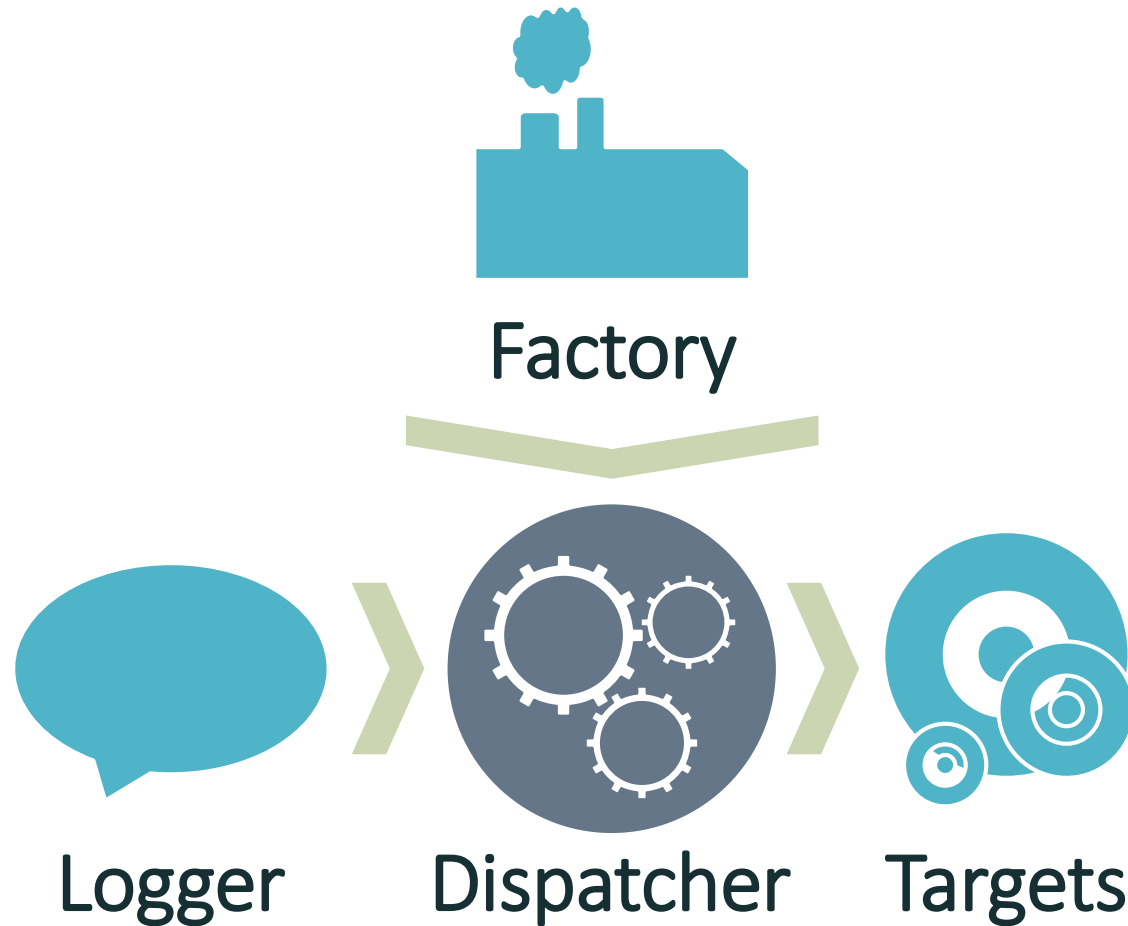
Rules

- Rules define which targets a log event is dispatched to using the assigned tags
- Rules can be defined with:
 - Which tags to include in the rule
 - Which tags to exclude in the rule (specifically from those selected by "include")
 - Whether or not all tags defined as "include" must be matched by the log event to match the rule
 - Which targets matching log events are to be dispatched to
 - Whether or not to process any further rules for the log event if the current rule matches



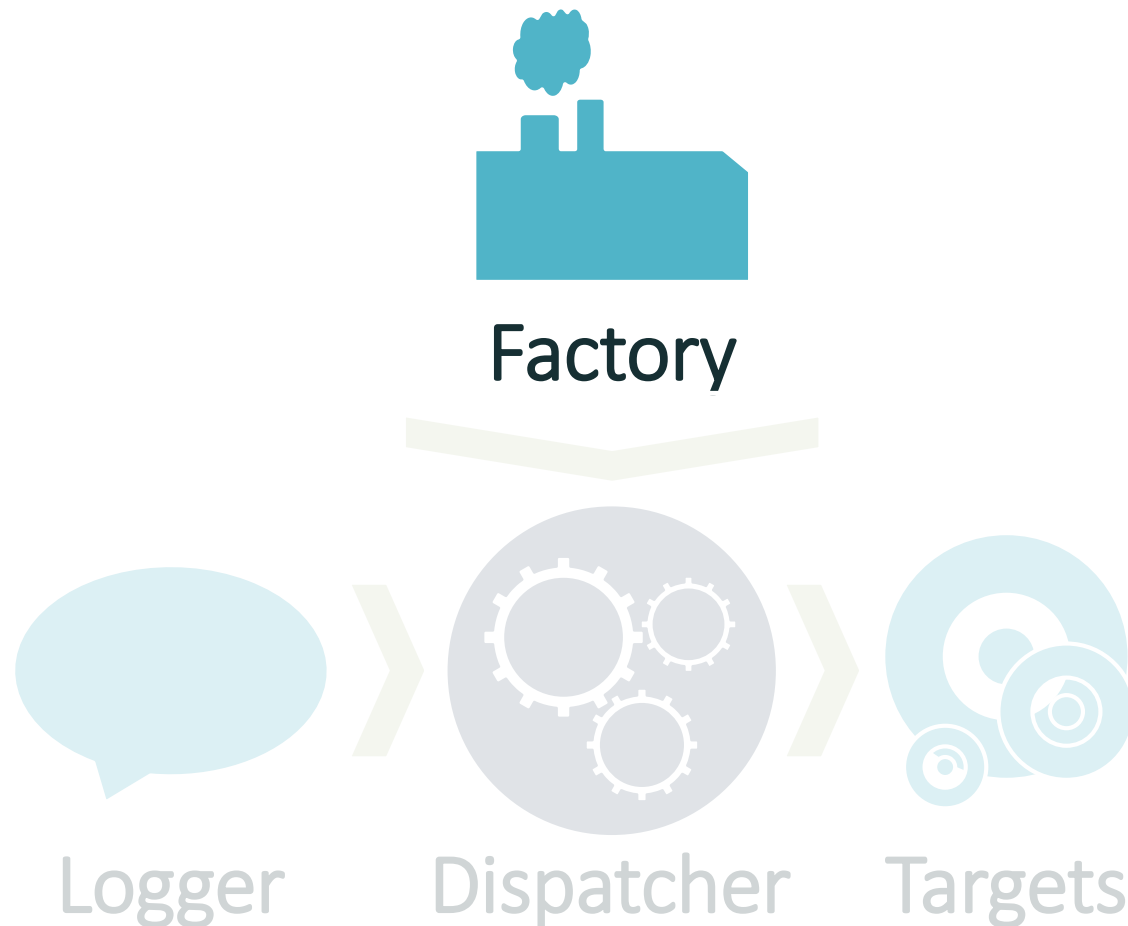
Basic Architecture

Coming together is a beginning; keeping together is progress; working together is success



Modular Architecture

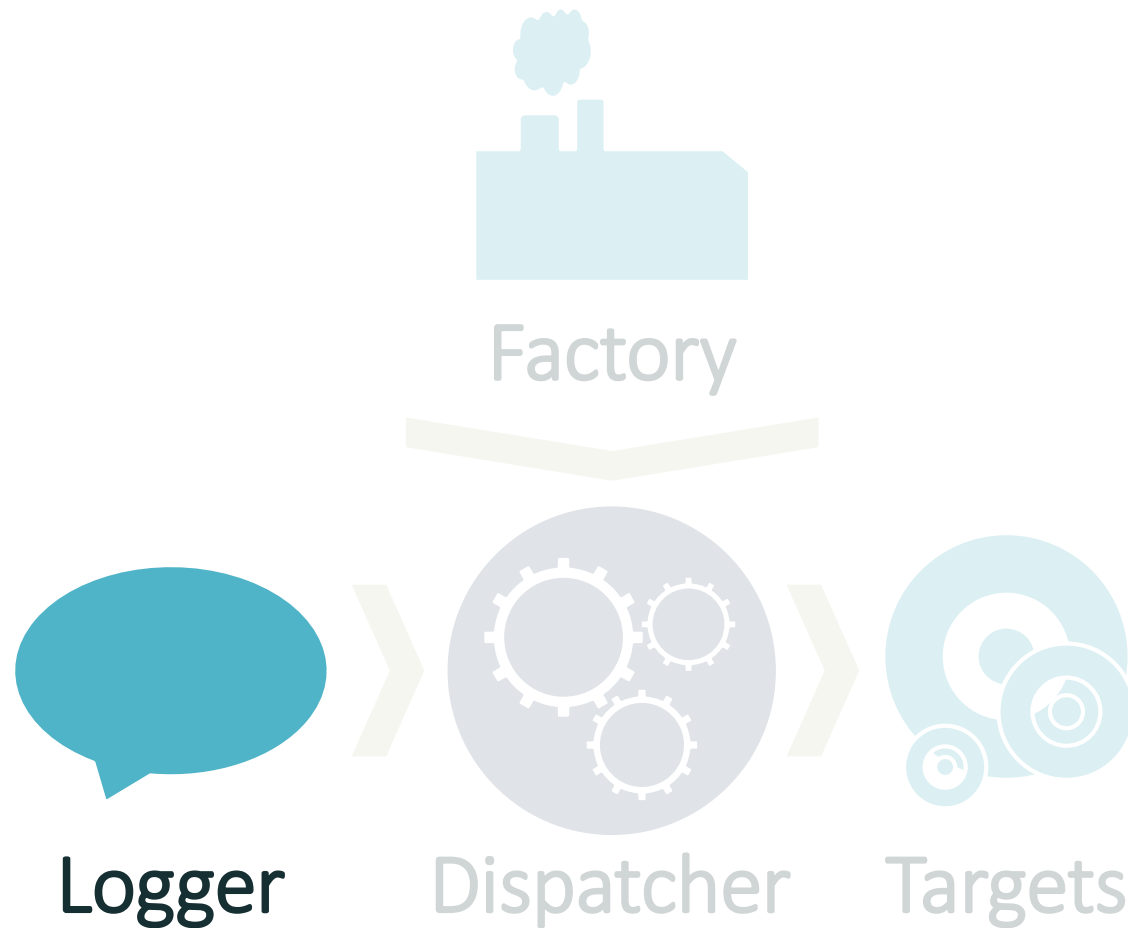
- The framework is divided into four major parts:
 - A factory for building the components of the framework
 - Loggers for sending log events to the dispatcher
 - A Dispatcher for dispatching log events to the log targets based on the configuration
 - Targets to receive and handle dispatched log events



Factory

- The Factory:
 - Reads/receives, builds, manages and distributes the configuration
 - Builds the dispatcher and targets using the configuration¹
 - Builds and provides loggers to the implementing application
 - Shuts down dispatcher and targets when the application is finished

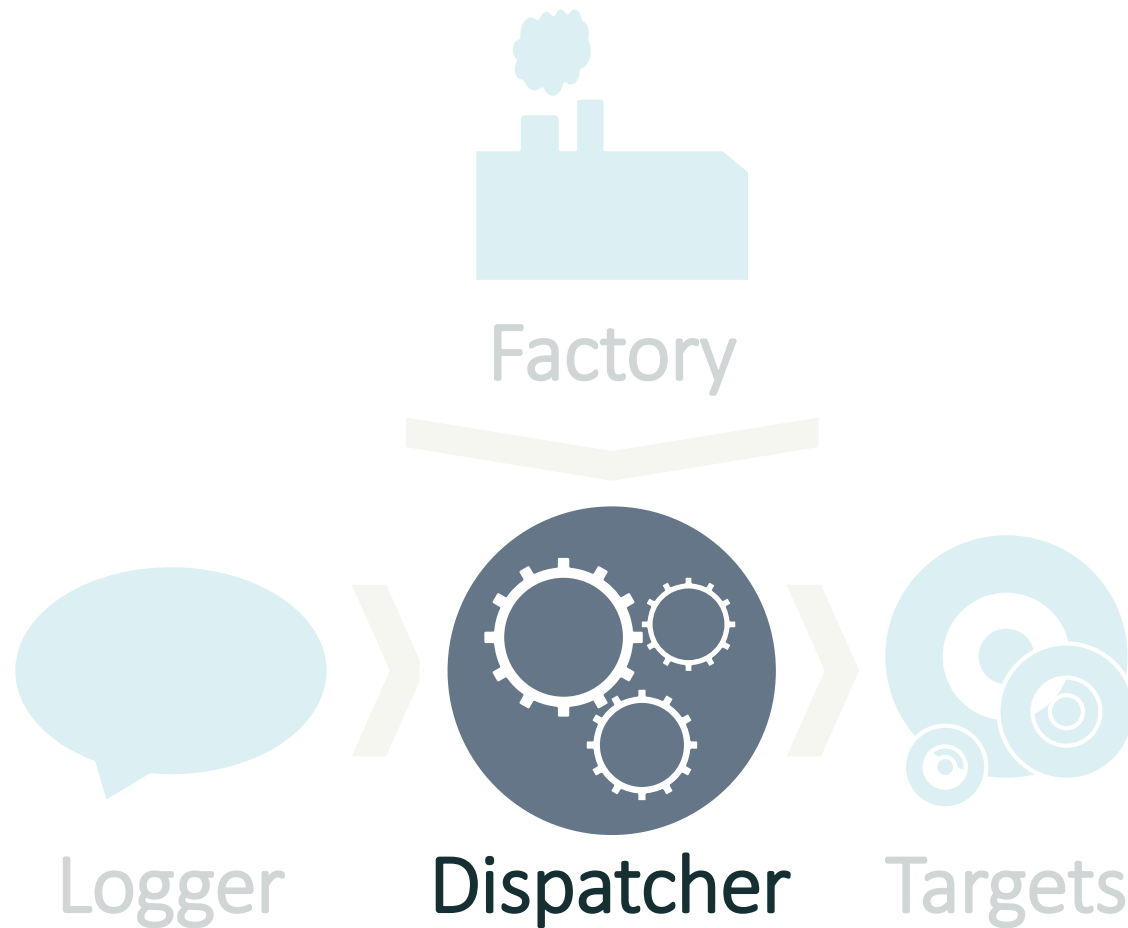
1. The factory leverages the Singleton Pattern to help keep resource usage low



Logger

- A logger:
 - Is responsible for building and sending log events to the dispatcher
 - Provides “helper methods” for automatically assigning tags to log events
 - Provides “helper methods” for automatically assigning meta data to log events
 - Can easily be extended to provide other “helper methods”¹

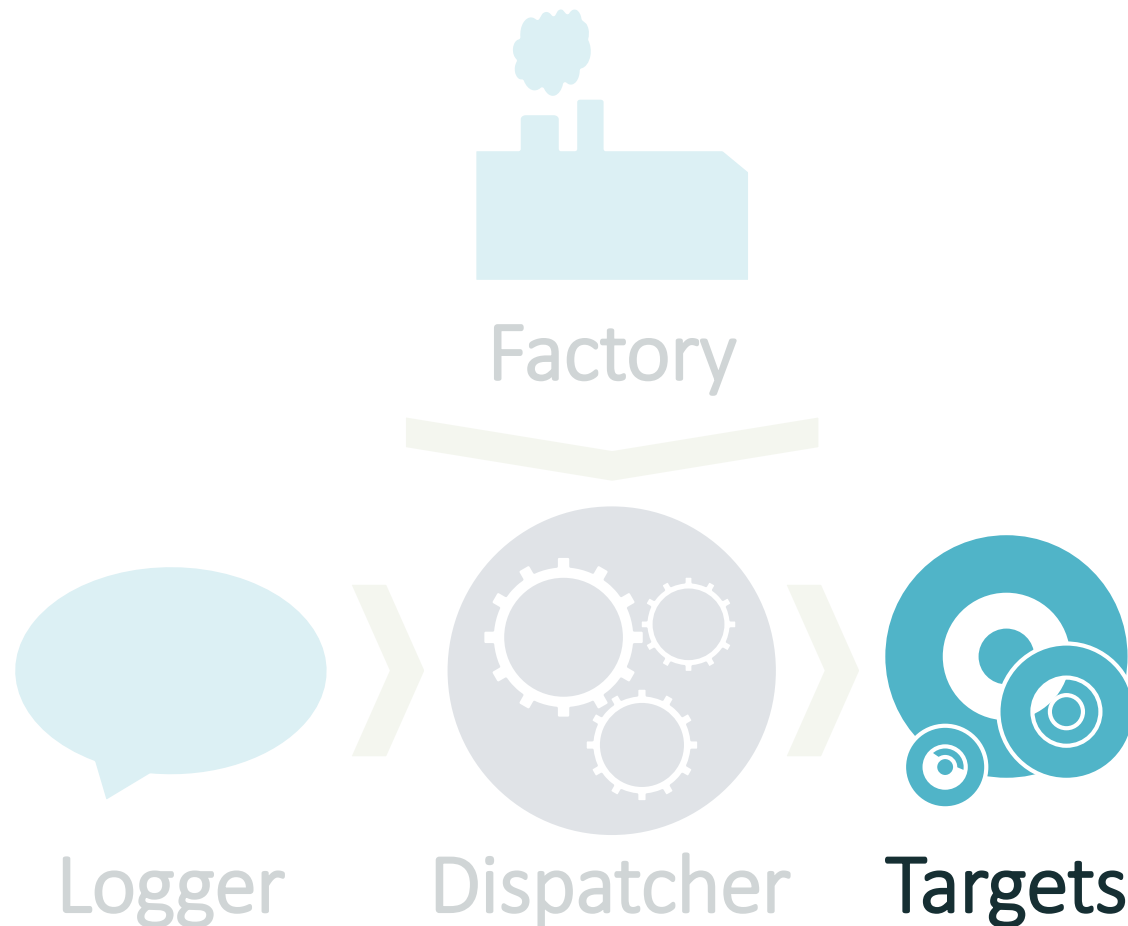
1. Using extension methods



Dispatcher

- The Dispatcher:
 - Is “The man behind the curtain”, invisible to the implementing application. It is responsible for dispatching log events to targets based on the configuration received from the factory
 - Allows for synchronous or asynchronous logging
 - Directly owns and is responsible for the log target instances (instantiation, notification and destruction)¹

1. Uses the Observer Pattern to manage Targets and dispatch log events



Targets

- A Target:
 - Is responsible for receiving, interpreting and processing log events
 - Receives log events from the dispatcher¹
 - Handles the log event, converting and formatting it for delivery
 - Delivers the log event to the final destination, such as to the screen or a file
 - Also has some concurrency logic to improve performance and handling of log events²

1. The dispatcher uses the Observer Pattern to manage Targets and dispatch log events, where the Dispatcher is the subject and the targets are the observers

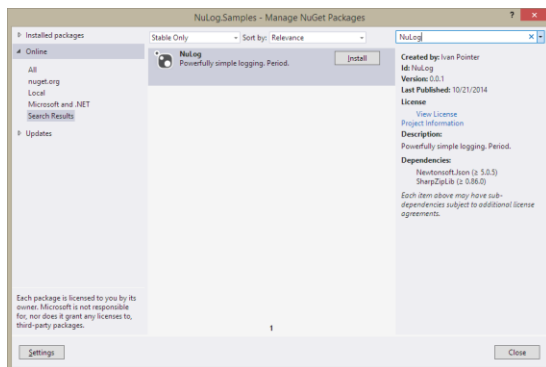
2. The architecture of synchronous vs. asynchronous logging is carefully considered, minimizing the complexity of these implementations for developers making custom targets



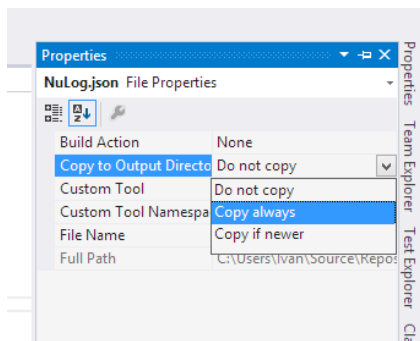
Implementing NuLog

Everything should be as simple as it is, but not simpler

1



2



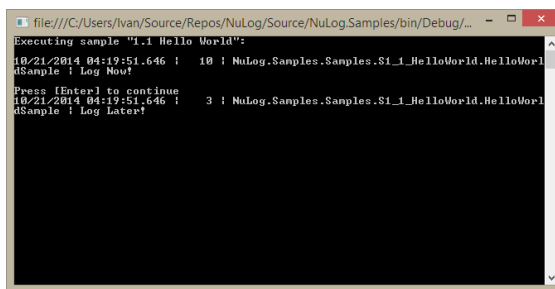
3

```
using NuLog;

namespace NuLog.Samples.Samples.S1_1_HelloWorld
{
    /// <summary> ...
    class HelloWorldSample : SampleBase
    {
        // The logger
        private static readonly LoggerBase _logger = LoggerFactory.GetLogger();

        // Logging example
        public override void ExecuteSample(Arguments args)
        {
            _logger.Log("Log Later!");
            _logger.LogNow("Log Now!");
        }
    }
}
```

4



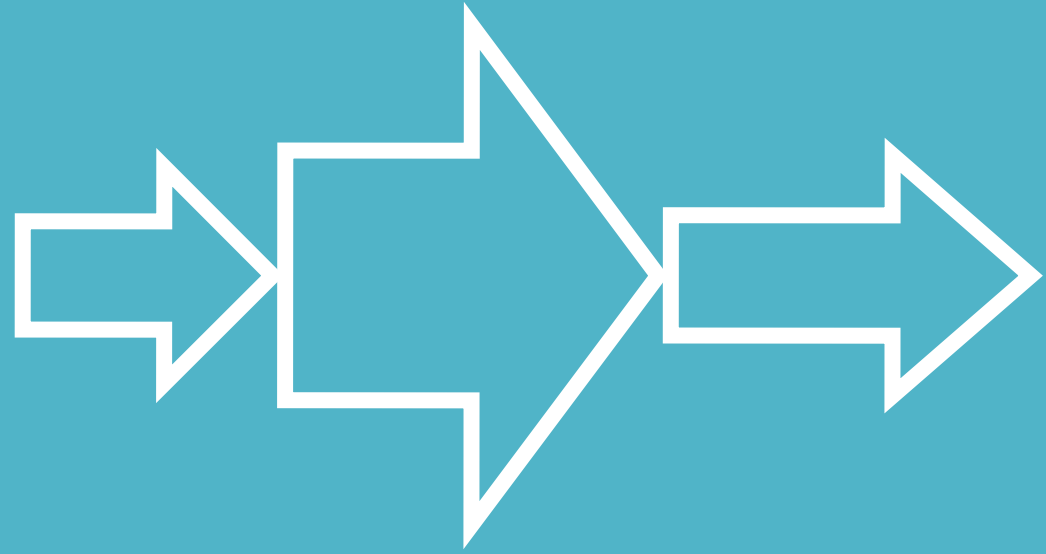
Project Setup

1. Install the NuLog package using NuGet
2. Mark the NuLog.json file for export, and edit it for your specific needs¹
3. “Use” the NuLog package, get an instance of the logger, then use it!
4. Enjoy the benefits of NuLog!

1. More on the details of configuration later in this presentation



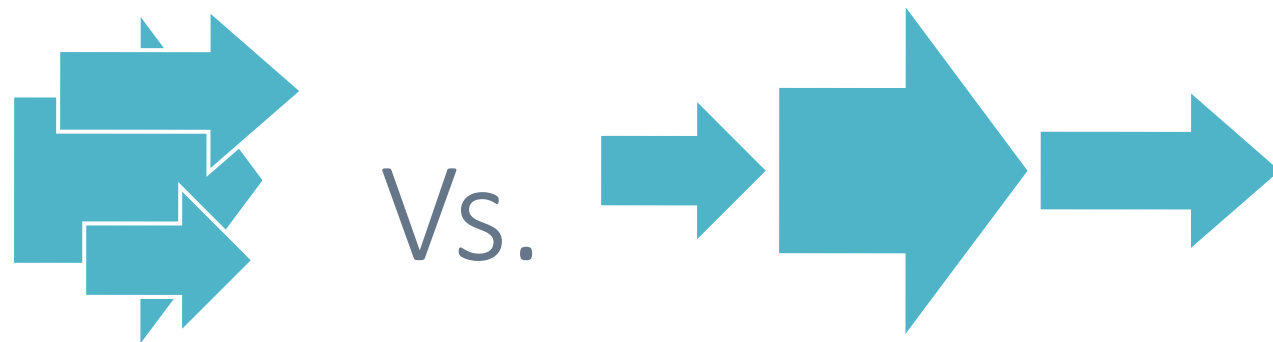
Vs.



Asynchronous Vs. Synchronous Logging

Do three things well, not ten things badly

Asynchronous Vs. Synchronous Logging



- Log events can be dispatched either now, or later
- When to send a log event can be decided when the logger is called, in the configuration, or by the target¹

1. Targets can pre-determine whether or not they are synchronous; for example, the console target defaults to send log events synchronously. This can be overridden in the configuration, assuming the target honors this configuration. Messages are still sent using a separate thread when using the asynchronous option of the logger however, but the log events maintain their order and are logged synchronously by the target.

```
var logger = LoggerFactory.GetLogger();

logger.Log("I am logging asynchronously");

logger.LogNow("I am logging right now!");

{
  "targets": [
    {
      "name": "console",
      "type": "NuLog.Targets.ConsoleTarget",
      "synchronous": true
    }
  ],
  "rules": [
    {
      "include": [ "*" ],
      "writeTo": [ "console" ],
      "final": false
    }
  ],
  "synchronous": true,
  "watch": true
}
```

Concurrent Logging in the Logger

- The base logger provides two types of log functions:
 - logger.Log will hint to the framework that this message can be sent asynchronously, unless the target configuration is set to send synchronously
 - logger.LogNow will hint to the framework that this message needs to be sent synchronously
 - If a log event is sent synchronously using "LogNow", control will not be returned to the application until the log event has been dispatched to the targets
 - This behavior can be overridden at the target, or global configuration level

```
var logger = LoggerFactory.GetLogger();

logger.Log("I am logging asynchronously");

logger.LogNow("I am logging right now!");

{
  "targets": [
    {
      "name": "console",
      "type": "NuLog.Targets.ConsoleTarget",
      "synchronous": true
    }
  ],
  "rules": [
    {
      "include": [ "*" ],
      "writeTo": [ "console" ],
      "final": false
    }
  ],
  "synchronous": true,
  "watch": true
}
```

Concurrent Logging in the Target

- A target can be set to synchronous logging using its configuration:
 - If “Log” is used, control is immediately returned to the calling application, but the log event is passed to the targets in another thread in a synchronous manner
- A target can be set to synchronous in its implementation:
 - The architecture of the target and its configuration are designed such that a target can default to be synchronous. This is designed however, to be “overridden” in the configuration, either way.

```
var logger = LoggerFactory.GetLogger();

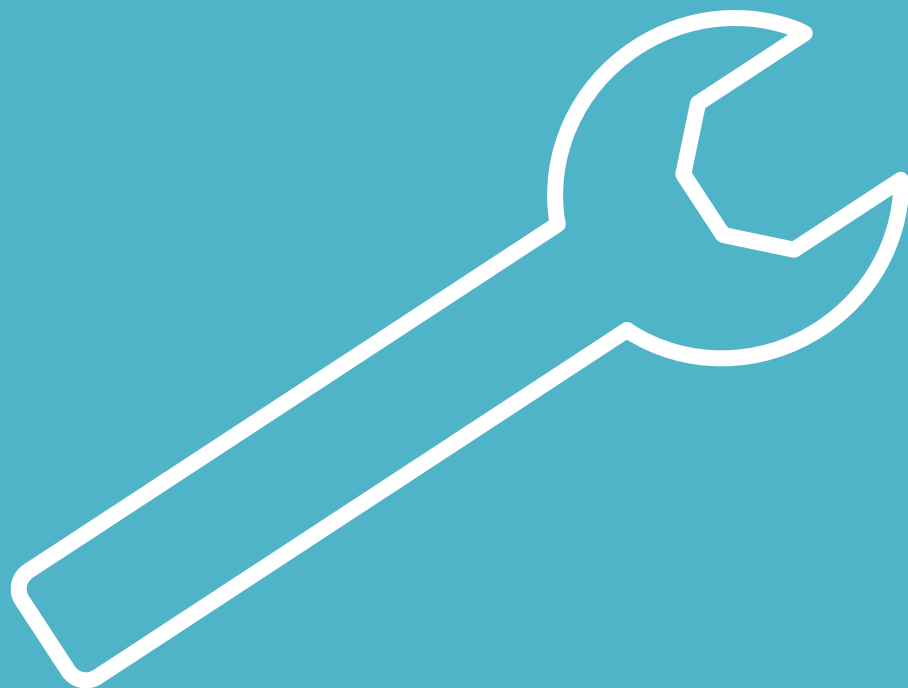
logger.Log("I am logging asynchronously");

logger.LogNow("I am logging right now!");

{
  "targets": [
    {
      "name": "console",
      "type": "NuLog.Targets.ConsoleTarget",
      "synchronous": true
    }
  ],
  "rules": [
    {
      "include": [ "*" ],
      "writeTo": [ "console" ],
      "final": false
    }
  ],
  "synchronous": true,
  "watch": true
}
```

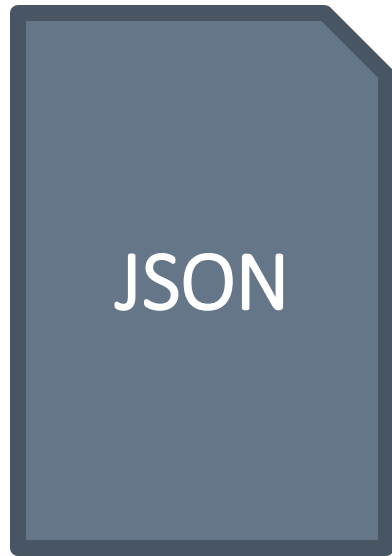
Disabling Concurrent Logging

- Concurrent logging can be disabled at a “global” level in the configuration:
 - If “synchronous” is set to true at the base configuration level, asynchronous logging will be completely turned off; all log events will be handled synchronously and no additional threads will be spun up to handle the logging; all logging will be performed on the thread that owns the framework.

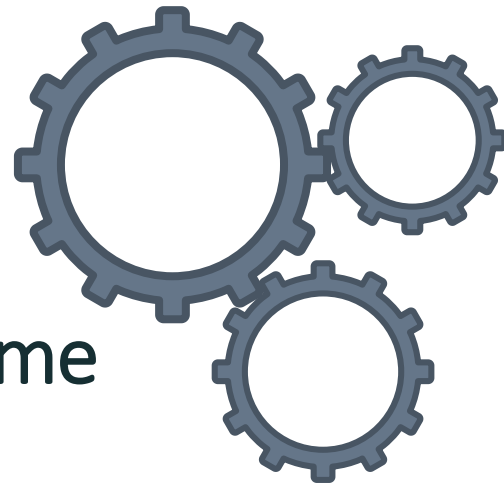


Configuration

Everything should be as simple as it is, but not simpler



File



Runtime

Configuration

- NuLog can be configured using a JSON configuration file, or:
- NuLog can be configured using a runtime configuration.
- NuLog is designed to re-configure itself at runtime when the configuration changes¹

1. NuLog uses the Observer Pattern to manage and announce configuration changes



JSON

File

```
{
  "targets": [
    {
      "name": "console",
      "type": "NuLog.Targets.ConsoleTarget"
    },
    {
      "name": "trace",
      "type": "NuLog.Targets.TraceTarget"
    }
  ],
  "rules": [
    {
      "include": [ "NuLog.Samples.Samples.S1_2_TagBasics.*" ],
      "writeTo": [ "trace" ],
      "final": false
    },
    {
      "include": [ "mytag" ],
      "writeTo": [ "console" ],
      "final": false
    }
  ],
  "watch": true
}
```

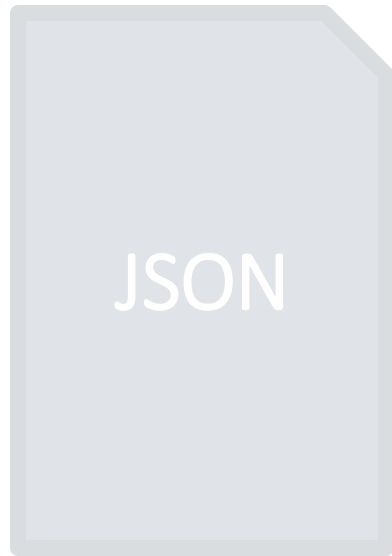


Runtime

File Configuration

- Configuration files are JSON files (*.json)¹
- A single configuration file contains the configuration for all of the logging
- Using configuration files allows NuLogging to monitor changes to the file, allowing for logging to be updated at runtime via the file
- The rest of this presentation will use the JSON configuration format to sample the configuration structure

1. The default name for the configuration file is 'NuLog.json' and is assumed to be in the "working directory" of the implementing application. If a file with a different name or location is used, it must be specified to the factory at runtime, or in the application config setting "NuLog.File".



File

Runtime Configuration

- Runtime configuration allows for other methods of configuring the logging via extension; such as using web or app config using a custom adapter
- Helper classes called “Configuration Builders” are also provided to help with building the configuration at runtime¹

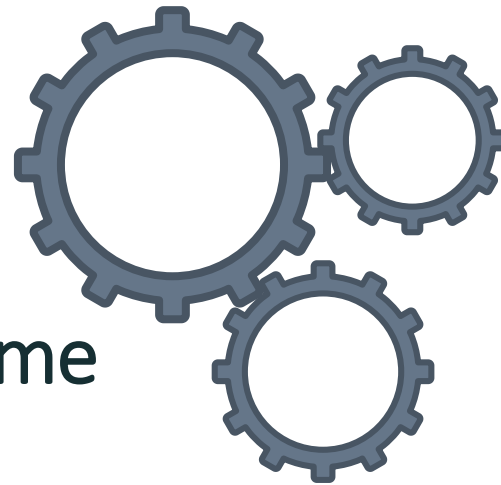
```
var config = LoggingConfigBuilder.CreateLoggingConfig()
    .AddTarget
    (
        ConsoleTargetConfigBuilder.Create()
            .SetLayoutConfig(new LayoutConfig("${Message}"))
            .Build()
    )
    .Build();

LoggerFactory.Initialize(config);

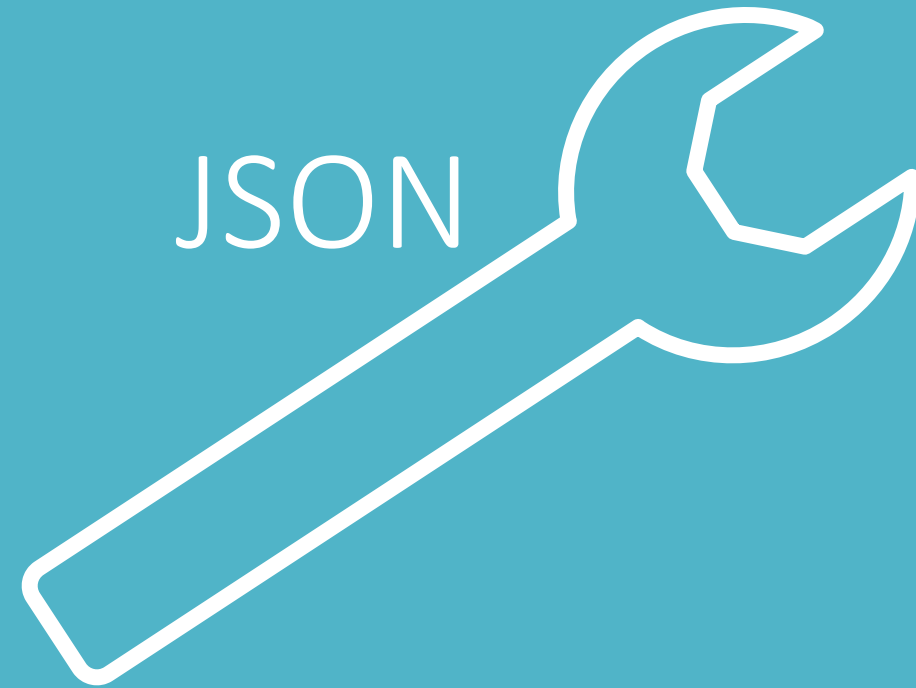
var logger = LoggerFactory.GetLogger();

logger.Log("Hello, World!");
```

Runtime



1. The configuration builders use the Factory Method and Method Chaining patterns to help with building the configuration objects. If you create a custom target, you should provide “Configuration Builders” as well. A separate presentation will be provided about extending NuLog.



JSON Configuration

Everything should be as simple as it is, but not simpler

```

{
  "targets": [
    {
      "name": "console",
      "type": "NuLog.Targets.ConsoleTarget"
    },
    {
      "name": "trace",
      "type": "NuLog.Targets.TraceTarget"
    }
  ],
  "rules": [
    {
      "include": [ "NuLog.Samples.Samples.S1_2_TagBasics.*" ],
      "writeTo": [ "trace" ],
      "final": false
    },
    {
      "include": [ "mytag" ],
      "writeTo": [ "console" ],
      "final": false
    }
  ],
  "tagGroups": {
    "fruit": [ "banana", "orange", "strawberry", "tomato", "grape" ],
    "animals": [ "dog", "cat", "fish", "giraffe" ],
    "errors": [ "exception", "failure", "fail", "fault" ]
  },
  "debug": false,
  "watch": true,
  "synchronous": false
}

```

Configuration Anatomy

- The logging configuration consists of 5 major pieces:
 - Targets
 - Rules
 - Tag Groups
 - Debug Flag
 - Watch Flag¹
 - Synchronous Flag

1. The watch flag is exclusive to the JSON file based configuration. More on this later in this presentation.

```

{
  "targets": [
    {
      "name": "console",
      "type": "NuLog.Targets.ConsoleTarget"
    },
    {
      "name": "trace",
      "type": "NuLog.Targets.TraceTarget"
    }
  ],
  "rules": [
    {
      "include": [ "NuLog.Samples.Samples.S1_2_TagBasics.*" ],
      "writeTo": [ "trace" ],
      "final": false
    },
    {
      "include": [ "mytag" ],
      "writeTo": [ "console" ],
      "final": false
    }
  ],
  "tagGroups": {
    "fruit": [ "banana", "orange", "strawberry", "tomato", "grape" ],
    "animals": [ "dog", "cat", "fish", "giraffe" ],
    "errors": [ "exception", "failure", "fail", "fault" ]
  },
  "debug": false,
  "watch": true,
  "synchronous": false
}

```

Targets

- “targets” define the different targets that log messages can be dispatched to

```

{
  "targets": [
    {
      "name": "console",
      "type": "NuLog.Targets.ConsoleTarget"
    },
    {
      "name": "trace",
      "type": "NuLog.Targets.TraceTarget"
    }
  ],
  "rules": [
    {
      "include": [ "NuLog.Samples.Samples.S1_2_TagBasics.*" ],
      "writeTo": [ "trace" ],
      "final": false
    },
    {
      "include": [ "mytag" ],
      "writeTo": [ "console" ],
      "final": false
    }
  ],
  "tagGroups": {
    "fruit": [ "banana", "orange", "strawberry", "tomato", "grape" ],
    "animals": [ "dog", "cat", "fish", "giraffe" ],
    "errors": [ "exception", "failure", "fail", "fault" ]
  },
  "debug": false,
  "watch": true,
  "synchronous": false
}

```

Rules

- “rules” define which log events are dispatched to which targets


```

{
  "targets": [
    {
      "name": "console",
      "type": "NuLog.Targets.ConsoleTarget"
    },
    {
      "name": "trace",
      "type": "NuLog.Targets.TraceTarget"
    }
  ],
  "rules": [
    {
      "include": [ "NuLog.Samples.Samples.S1_2_TagBasics.*" ],
      "writeTo": [ "trace" ],
      "final": false
    },
    {
      "include": [ "mytag" ],
      "writeTo": [ "console" ],
      "final": false
    }
  ],
  "tagGroups": {
    "fruit": [ "banana", "orange", "strawberry", "tomato", "grape" ],
    "animals": [ "dog", "cat", "fish", "giraffe" ],
    "errors": [ "exception", "failure", "fail", "fault" ]
  },
  "debug": false,
  "watch": true,
  "synchronous": false
}

```

Tag Groups

- “tagGroups” define groups of tags which assist in defining rules¹

1. This is an advanced feature and will be covered in more detail later in this presentation

```

{
  "targets": [
    {
      "name": "console",
      "type": "NuLog.Targets.ConsoleTarget"
    },
    {
      "name": "trace",
      "type": "NuLog.Targets.TraceTarget"
    }
  ],
  "rules": [
    {
      "include": [ "NuLog.Samples.Samples.S1_2_TagBasics.*" ],
      "writeTo": [ "trace" ],
      "final": false
    },
    {
      "include": [ "mytag" ],
      "writeTo": [ "console" ],
      "final": false
    }
  ],
  "tagGroups": {
    "fruit": [ "banana", "orange", "strawberry", "tomato", "grape" ],
    "animals": [ "dog", "cat", "fish", "giraffe" ],
    "errors": [ "exception", "failure", "fail", "fault" ]
  },
  "debug": false,
  "watch": true,
  "synchronous": false
}

```

Debug

- “debug” instructs the system to include the calling method information in every log message, instead of just ones with exceptions

```

{
  "targets": [
    {
      "name": "console",
      "type": "NuLog.Targets.ConsoleTarget"
    },
    {
      "name": "trace",
      "type": "NuLog.Targets.TraceTarget"
    }
  ],
  "rules": [
    {
      "include": [ "NuLog.Samples.Samples.S1_2_TagBasics.*" ],
      "writeTo": [ "trace" ],
      "final": false
    },
    {
      "include": [ "mytag" ],
      "writeTo": [ "console" ],
      "final": false
    }
  ],
  "tagGroups": {
    "fruit": [ "banana", "orange", "strawberry", "tomato", "grape" ],
    "animals": [ "dog", "cat", "fish", "giraffe" ],
    "errors": [ "exception", "failure", "fail", "fault" ]
  },
  "debug": false,
  "watch": true,
  "synchronous": false
}

```

Watch

- “**watch**” tells the factory to watch the file for changes, allowing for runtime changes to configuration¹

1. The watch flag is exclusive to the JSON file based configuration. Note that if the developer sets “watch” to false at runtime, the file will no longer be watched by the system, and to turn file watching back on, the flag will need to be set to “true” and the implementing application restarted

```

{
  "targets": [
    {
      "name": "console",
      "type": "NuLog.Targets.ConsoleTarget"
    },
    {
      "name": "trace",
      "type": "NuLog.Targets.TraceTarget"
    }
  ],
  "rules": [
    {
      "include": [ "NuLog.Samples.Samples.S1_2_TagBasics.*" ],
      "writeTo": [ "trace" ],
      "final": false
    },
    {
      "include": [ "mytag" ],
      "writeTo": [ "console" ],
      "final": false
    }
  ],
  "tagGroups": {
    "fruit": [ "banana", "orange", "strawberry", "tomato", "grape" ],
    "animals": [ "dog", "cat", "fish", "giraffe" ],
    "errors": [ "exception", "failure", "fail", "fault" ]
  },
  "debug": false,
  "watch": true,
  "synchronous": false
}

```

Synchronous

- “synchronous”, when set to **true** tells the framework to handle all log events synchronously; no additional threads will be spun up for logging; all logging will be performed on the thread that owns the logging framework instance



Runtime Configuration

Everything should be as simple as it is, but not simpler

```
var config = LoggingConfigBuilder.CreateLoggingConfig()
    .AddTarget
    (
        ConsoleTargetConfigBuilder.Create()
            .SetLayoutConfig(new LayoutConfig("${Message}"))
            .Build()
    )
    .Build();

LoggerFactory.Initialize(config);

var logger = LoggerFactory.GetLogger();

logger.Log("Hello, World!");
```

Runtime Configuration

- The entire logging configuration is represented by a `LoggingConfig` instance
- “Configuration Builders” leverage the “Factory Method” and “Method Chaining” patterns to simplify building configurations at runtime
- The configuration keeps the same structure as the JSON configuration, allowing for the targets to be implemented agnostic to where configuration came from
- The `LoggerFactory` allows initializing with a runtime built configuration object¹

1. This is the standard use of runtime configuration and the factory should be initialized before any logger instances are established

```
var config = LoggingConfigBuilder.CreateLoggingConfig()
    .AddTarget
    (
        ConsoleTargetConfigBuilder.Create()
            .SetLayoutConfig(new LayoutConfig("${Message}"))
            .Build()
    )
    .Build();

LoggerFactory.Initialize(config);

var logger = LoggerFactory.GetLogger();

logger.Log("Hello, World!");
```

Runtime Configuration

- The Factory uses the observer pattern for managing the configuration; the configuration can be updated and the factory re-initialized (Using `LoggerFactory.Initialize`), which will notify all of the dependent objects of the new configuration
- When using runtime configuration, loggers need to be retrieved **after** the factory has been initialized with a configuration. They can thereafter be used as normal
- The Factory caches instances of the logger in a singleton-like pattern, minimizing the overhead of getting a logger multiple times in a single instance¹

1. Unless a runtime Meta Data Provider instance is provided, at which point, a new instance of the logger is created. More on Meta Data Providers later in this presentation.

```
var config = LoggingConfigBuilder.CreateLoggingConfig()
    .AddTarget
    (
        ConsoleTargetConfigBuilder.Create()
            .SetLayoutConfig(new LayoutConfig("${Message}"))
            .Build()
    )
    .Build();

LoggerFactory.Initialize(config);

var logger = LoggerFactory.GetLogger();

logger.Log("Hello, World!");
```

Runtime Configuration

- Runtime configuration is used primarily for advanced implementations, such as extensions or wrappers¹

1. An organization could use these features to wrap the framework to fit the organization's particular needs, while at the same time, minimizing complexity for their implementing developers



Standard Targets

Talent hits a target no one else can hit; Genius hits a target no one else can see



```
{  
  "name": "basic",  
  "type": "NuLog.Targets.TargetBase",  
  "synchronous": false  
}
```

Basic Target (Abstract)

- All targets must extend the basic target
- The basic target has a name, type and a synchronous flag:
 - The name identifies the target to the rules
 - The type is the full name of the class of the target¹
 - The synchronous flag is optional and defaults to false. If synchronous is set to true, log events are always sent directly to this target as opposed to being handled asynchronously²

1. Reflection is used to construct the target. When using custom targets in other assemblies, the assembly must be listed in the "type" property as well.

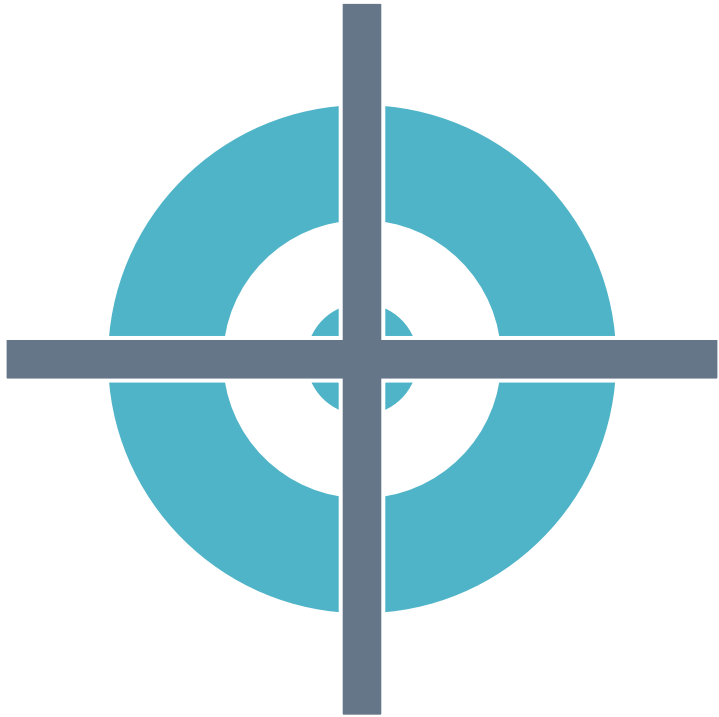
2. Assuming the target is built to honor this configuration. More on asynchronous vs. synchronous logging later in this presentation.



```
{  
  "name": "layout",  
  "type": "NuLog.Targets.LayoutTargetBase",  
  "layout": "${DateTime:'{0:MM/dd/yyyy hh:mm:ss.fff}'} | ${Thread.ManagedThreadId:'{0,4}'} | ${Tags} | ${Message}${?Exception:'\r\n{0}'}\r\n",  
  "synchronous": false  
}
```

Layout Target (Abstract)

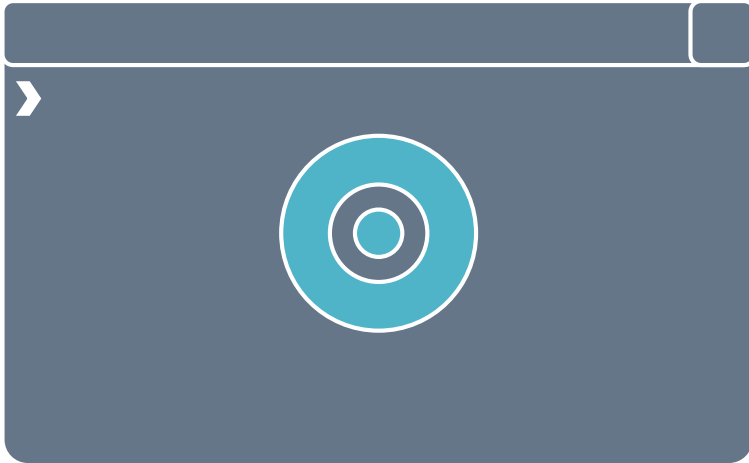
- The layout target defines a standard method for formatting the layout of text-based targets with a single layout format
- The Layout Target is used for the standard text-based targets
- Layouts are explored further, later in this presentation



```
{  
  "name": "trace",  
  "type": "NuLog.Targets.TraceTarget"  
}
```

Trace Target

- Built from the Layout Target
- The most basic of the standard targets
- Writes to `System.Diagnostics.Trace`

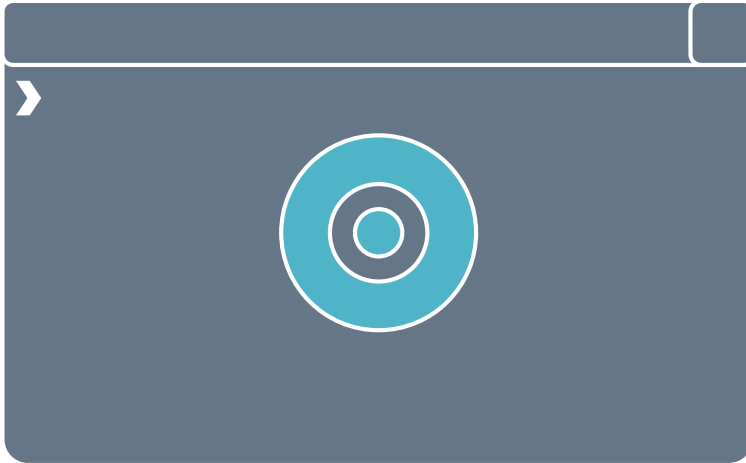


```
{  
  "name": "console",  
  "type": "NuLog.Targets.SimpleConsoleTarget"  
}
```

Simple Console Target

- Built from the Layout Target
- Logs to the console using `Console.Write`¹
- Simple, but high performance

1. Because `Write` is used (instead of `WriteLine`), the layout needs to include a newline character at the end if desired



```
{
  "name": "console",
  "type": "NuLog.Targets.ConsoleTarget",
  "colorRules": [
    {
      "tags": [ "error" ],
      "foreground": "White",
      "background": "DarkRed"
    },
    {
      "tags": [ "warn" ],
      "foreground": "White",
      "background": "DarkYellow"
    }
  ]
}
```

Console Target

- Built from the Layout Target
- Logs to the console using `Console.Write`¹
- Administrators can define colors in the configuration which key off of tags. Colors are defined in `System.ConsoleColor`
- Colors can be defined specifically for a log event by setting `"ConsoleForeground"` and `"ConsoleBackground"` in the log event's meta data²
- The performance of this is slower than the "Simple Console Target", because this target handles coloring in the target

1. Because `Write` is used (instead of `WriteLine`), the layout needs to include a newline character at the end if desired

2. More on the meta data configuration later in this presentation



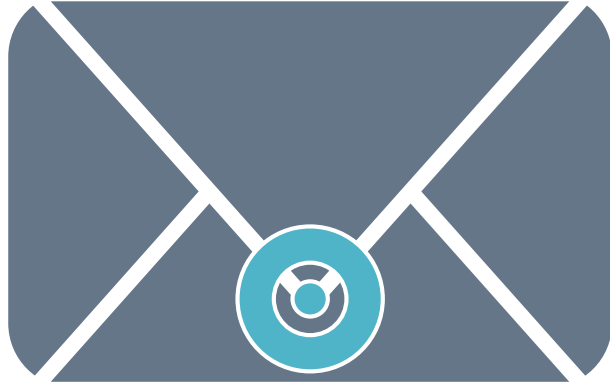
```
{  
  "name": "file",  
  "type": "NuLog.Targets.TextFileTarget",  
  "fileName": "test.log",  
  "oldFileNamePattern": "log{0:.yyyy.MM.dd.hh.mm.ss}.log",  
  
  "rolloverPolicy": "Size",  
  "rolloverTrigger": "10MB",  
  "oldFileLimit": 10,  
  "compressOldFiles": true,  
  "compressionLevel": 3,  
  "compressionPassword": "hellopasswd"  
}
```

"rolloverPolicy": "Day",
"oldFileLimit": 10,
"compressOldFiles": true,
"compressionLevel": 3,
"compressionPassword": "hellopasswd"

Text File Target

- Built from the Layout Target
- Logs to a specified text file
- Highly configurable¹
- Can rollover by size or daily
- Can compress and password-protect old log files

1. All options specific to the text file target are shown here, there are many optional options here that are not required for configuration. See the detailed specifications for the targets for more information.



```
{
  "name": "email",
  "type": "NuLog.Targets.EmailTarget",
  "host": "some.mail-relay.com",
  "port": 25,
  "userName": "smtpUser",
  "password": "smtpPassword",
  "enableSSL": "true",
  "fromAddress": "somebody@somewhere.org",
  "subject": "Fixed subject",
  "replyTo": "somebody@somewhere.org",
  "to": [ "somebody@somewhere.org" ],
  "cc": [ "somebodyelse@somewhere.org" ],
  "bcc": [ "secretsomebody@somewhere.org" ],
  "subjectLayout": "DEV ${Subject}",
  "bodyLayout": "${DateTime:'{0:MM/dd/yyyy hh:mm:ss.fff}'} | ${Thread.ManagedThreadId:'{0,4}'} | ${Tags} | ${Message}${?Exception:'\r\n{0}'}\r\n",
  "headers": {
    "extraheader": "headervalue"
  },
  "bodyFile": "bodyLayout.txt",
  "isBodyHtml": true
}
```

Email Target

- Built from the Layout Target
- Highly configurable¹
- Handles authentication and SSL
- Allows for adding “extra” headers to the email message
- Allows for defining the body layout in a text file
- Allows for sending attachments at runtime

1. All options specific to the email target are shown here, there are many optional options here that are not required for configuration. See the detailed specifications for the targets for more information.

$\$ \{ ? \text{Layouts} : ' \mid \{ 0, 4 \}' \}$

Layouts

Style is a simple way of saying complicated things

“My Property: \${?Property.Name:‘Formatted: {0}’}\r\n”

“Hello: \${Message}”

“Hello Layout\${?DateTime:': {0:MM/dd/yyyy hh:mm:ss.fff}'}!\r\n”

Layouts

- Layouts are a mechanism for converting a log event into text using a “layout” format
- Layouts are used by the standard text-based targets (and the SMTP target)
- Layouts allow for the formatting of different parts of the log event, even recursively
- Layouts allow for conditionally showing formatted parts of the log event

“Hello Layout\${?DateTime:': {0:MM/dd/yyyy hh:mm:ss.fff}'}!\r\n”

“Hello Layout: 10/13/2014 15:34:12.572!”

Static Text

- Anything not wrapped in a property enclosure \${} is treated as static text
- Static text will always show in a log event formatted by a layout
- Escaped characters are supported (and suggested!)

“Hello Layout\${?DateTime:' {0:MM/dd/yyyy hh:mm:ss.fff}'}!\r\n”



Parameter

- Parameters are wrapped with the property enclosure `${}`
- A single parameter in the layout format refers to a single property of the log event
- Parameters have 3 parts:
 - Conditional Flag (Optional)
 - Property Name (Required)
 - Property Format (Optional)

“Hello Layout\${?DateTime:'{0:MM/dd/yyyy hh:mm:ss.fff}'}!\r\n”

Conditional Flag
(Optional)

Property Name
(Required)

Property Format
(Optional)

Conditional Flag

- The conditional flag is a single ‘?’ located at the front of the property, inside the 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

“Hello Layout\${?DateTime:': {0:MM/dd/yyyy hh:mm:ss.fff}'}!\r\n”



Property Name

- The name of the property within the log event is located at the beginning of the property string, after the conditional flag
- All log events have optional “Meta Data”¹
- The Property Name value is reflective and recursive, child values can be accessed with periods, for example: DateTime.Day
- The “Meta Data” 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”

1. Meta Data is explored in more depth later in this presentation

“Hello Layout\${?DateTime:': {0:MM/dd/yyyy hh:mm:ss.fff}'}!\r\n”

Conditional Flag
(Optional)

Property Name
(Required)

Property Format
(Optional)

Property Format

- 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

`"Hello Layout${?DateTime:'': {0:MM/dd/yyyy hh:mm:ss.fff}}'\r\n"`



(Without DateTime Value)

`"Hello Layout!"`

(With DateTime Value)

`"Hello Layout: 10/13/2014 15:34:12.572!"`

Example

- In the example on the left:
 - The DateTime value in the log event is used as the value for the property
 - Without a DateTime value in the log event, the ":" in the string format would be excluded from the string because of the Conditional Flag
 - The DateTime is formatted using the format "{0:MM/dd/yyyy hh:mm:ss.fff}", as is specified in the layout format at the top

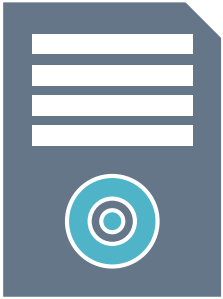
Email



Subject

Body

Text File



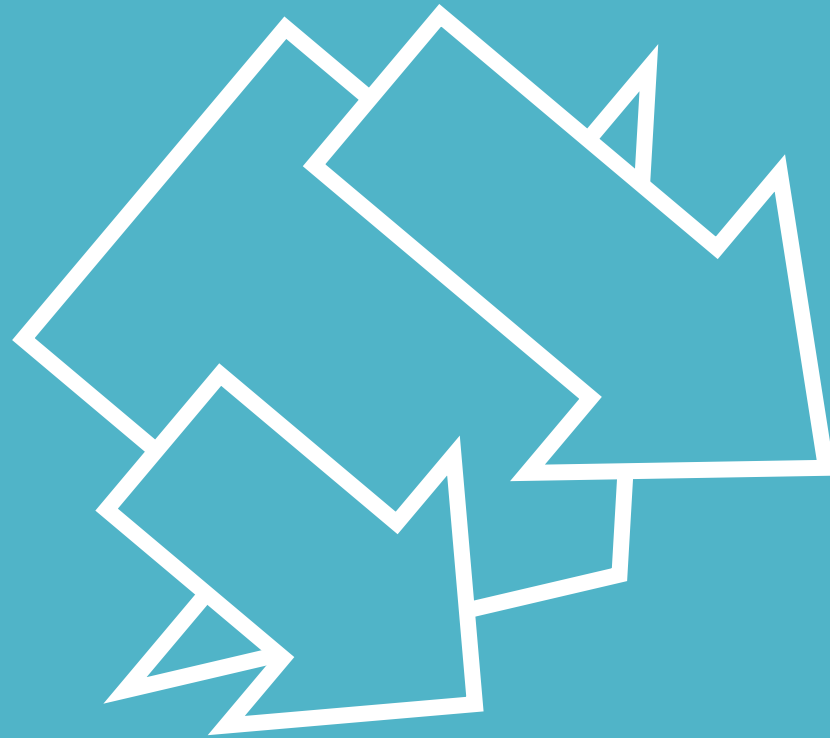
Trace



Simple Console/
Console

Layout in the Standard Targets

- Trace Target
- Console Target
- Text File Target
- Email Target
 - Subject line
 - Body



Rules

The golden rule is that there are no golden rules

#OID.Agency.Application.Class

```
"rules": [{  
  "include": [ "Some.Namespace.*" ],  
  "strictInclude": true,  
  "exclude": [ "Some.Namespace.Other.*", "trace" ],  
  "writeTo": [ "console" ],  
  "final": false  
}]
```

console

Rules

- Rules define which targets log events are dispatched to based on their assigned tags
- Rules are designed to work either independently, or in concert of each other
- Rules are defined with 5 properties: "include", "strictInclude", "exclude", "writeTo" and "final"

#OID.Agency.Application.Class

```
"rules": [{  
  "include": [ "Some.Namespace.*" ],  
  "strictInclude": true,  
  "exclude": [ "Some.Namespace.Other.*", "trace" ],  
  "writeTo": [ "console" ],  
  "final": false  
}]
```

console

Rules: Include

- “include” defines which tags to include in the rule
- Wildcards “*” can be used to broaden the scope of a tag entry¹
- If no tags are specified, all log events will be included in the rule

1. This is especially helpful for directing log events from a particular namespace or class to a target or set of targets

#OID.Agency.Application.Class

```
"rules": [{  
  "include": [ "Some.Namespace.*" ],  
  "strictInclude": true,  
  "exclude": [ "Some.Namespace.Other.*", "trace" ],  
  "writeTo": [ "console" ],  
  "final": false  
}]
```

console

Rules: Strict Include

- “strictInclude” identifies whether or not all tags identified by “include” must be matched by the log event for the log event to match the rule
- A value of “true” indicates that all tags defined in “include” must have a matching tag in the log event for the log event to match the rule

#OID.Agency.Application.Class

```
"rules": [{  
  "include": [ "Some.Namespace.*" ],  
  "strictInclude": true,  
  "exclude": [ "Some.Namespace.Other.*", "trace" ],  
  "writeTo": [ "console" ],  
  "final": false  
}]
```

console

Rules: Exclude

- “exclude” identifies which tags will be explicitly excluded by the rule
- Wildcards “*” can be used to broaden the scope of a tag entry¹
- If left blank, it is assumed that none of the log events identified by “include” will be excluded

1. This is especially helpful for preventing log events from a particular namespace or class to a target or set of targets

#OID.Agency.Application.Class

```
"rules": [{  
  "include": [ "Some.Namespace.*" ],  
  "strictInclude": true,  
  "exclude": [ "Some.Namespace.Other.*", "trace" ],  
  "writeTo": [ "console" ],  
  "final": false  
}]
```

console

Rules: Write To

- “writeTo” identifies which targets (by name) matching log events will be dispatched to

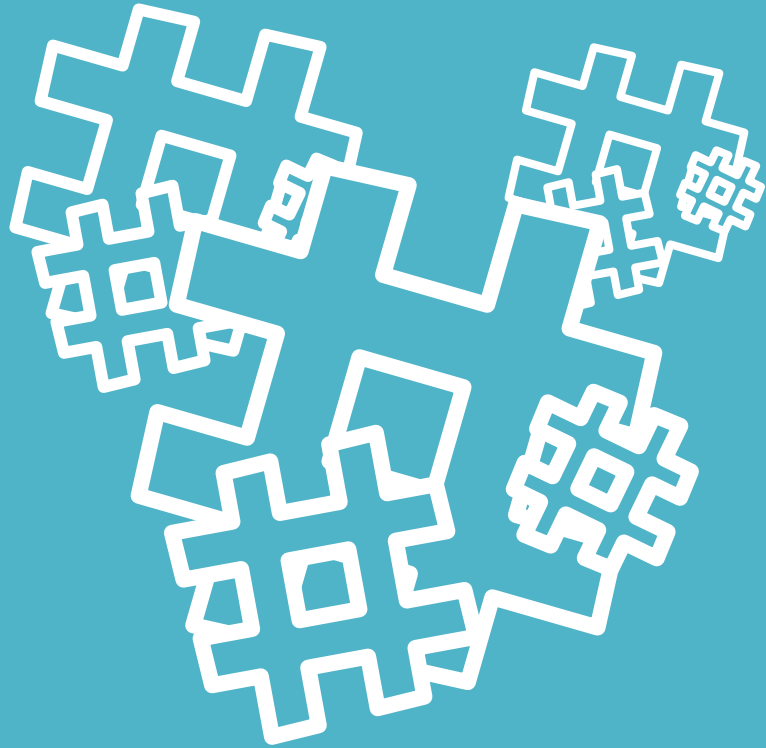
#OID.Agency.Application.Class

```
"rules": [{  
  "include": [ "Some.Namespace.*" ],  
  "strictInclude": true,  
  "exclude": [ "Some.Namespace.Other.*", "trace" ],  
  "writeTo": [ "console" ],  
  "final": false  
}]
```

console

Rules: Final

- “final” specifies whether or not any other rules will be processed if this rule matches the log event
- For example, if you have three rules that match a given log event, and the second is marked as “final”, the third will not be processed for the log event



Tag Groups

A small group of thoughtful people could change the world. Indeed, it's the only thing that ever has.

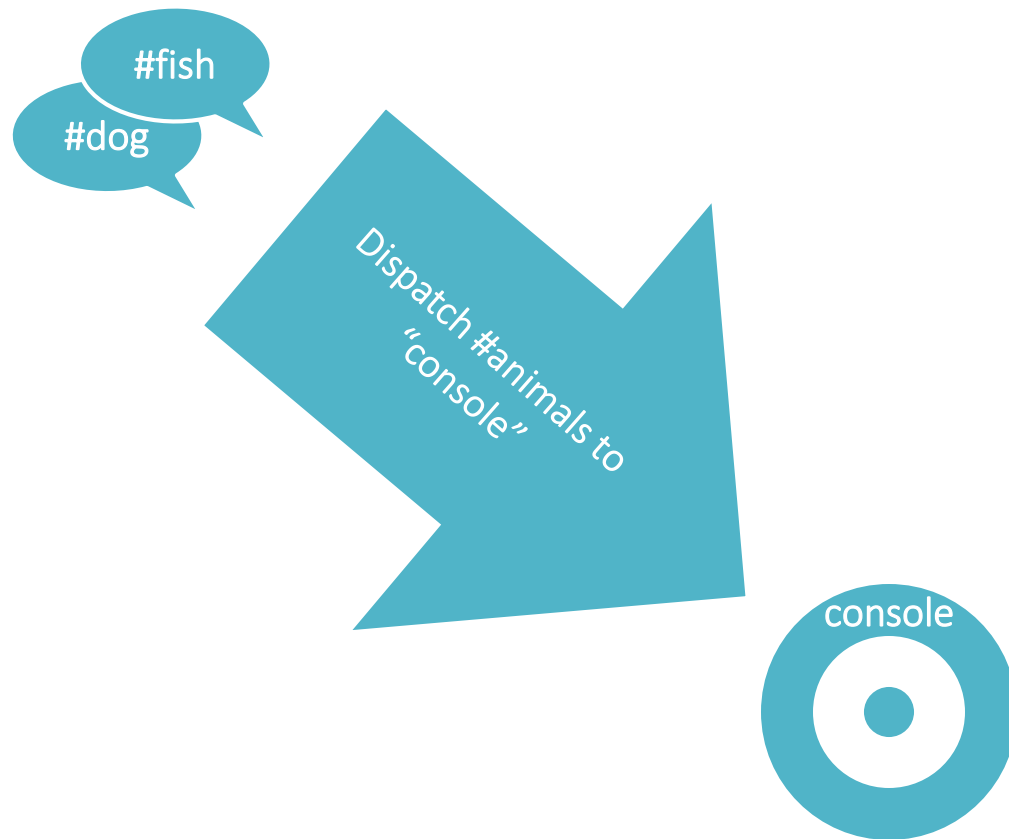
```
"tagGroups": {  
  "fruit": [ "banana", "orange", "strawberry", "tomato", "grape" ],  
  "animals": [ "dog", "cat", "fish", "giraffe" ],  
  "errors": [ "exception", "failure", "fail", "fault" ]  
},
```

Tag Groups

- Tag groups allow for grouping multiple tags under a single tag
- This is useful if you want a rule to apply for a “category” of tags, such as “fruit”, “animals” or “errors” (abstract example, but you can imagine how this could be helpful)¹
- Observing the JSON to the left, the member name is the tag that represents the group, the child array being the tags that belong to the group

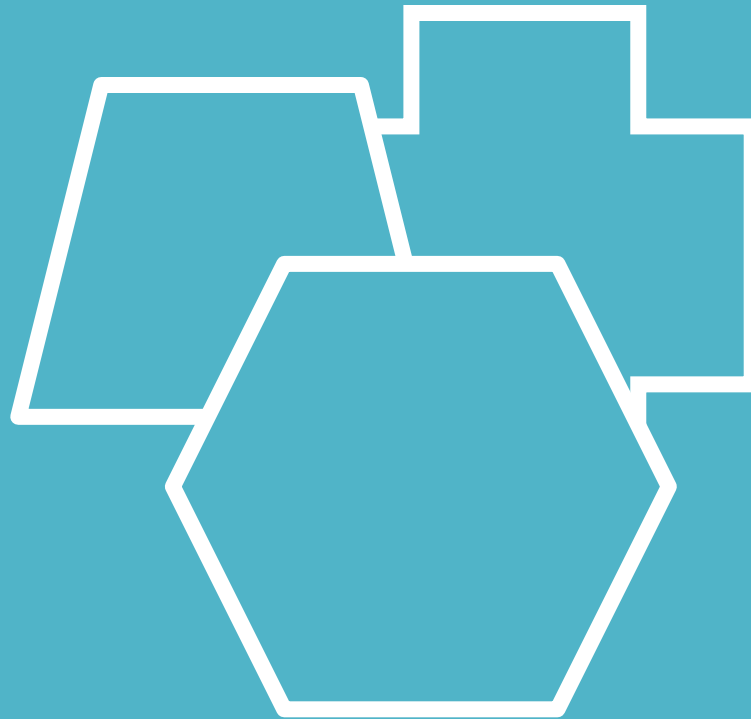
1. For example: the Legacy Logging extension uses Tag Groups to associate the different “trace” through “fatal” levels to each-other, allowing to emulate the legacy style of logging via levels

```
"tagGroups": {  
  "fruit": [ "banana", "orange", "strawberry", "tomato", "grape" ],  
  "animals": [ "dog", "cat", "fish", "giraffe" ],  
  "errors": [ "exception", "failure", "fail", "fault" ]  
},
```



Tag Groups

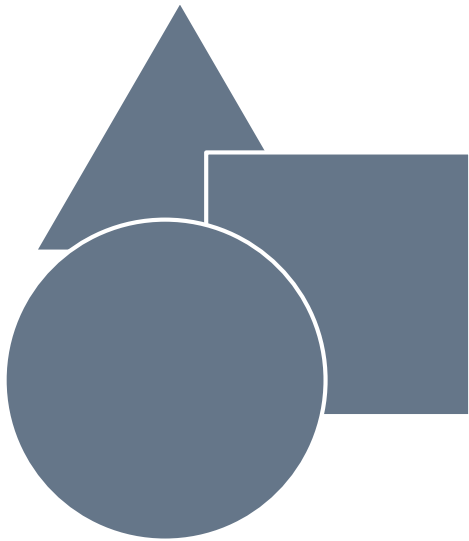
- For example:
 - Two log events, one tagged “fish” and one tagged “dog”
 - A single rule that states that “animals are to be dispatched to ‘console’”
 - Both “fish” and “dog” would be dispatched to the console



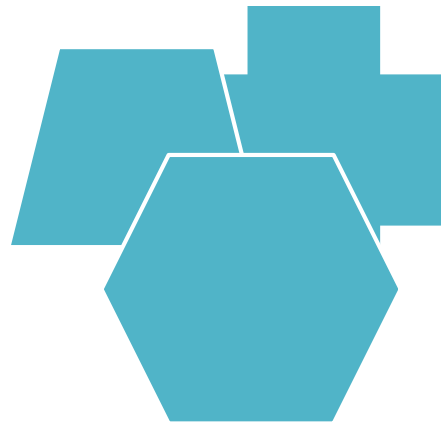
Meta Data

Growth is never by mere chance; it is the result of forces working together

Log Event



Meta Data



Meta Data


- Log events can have supplemental “Meta Data” which identifies additional information to be provided to the targets
- Meta Data stays with the log event and is therefore available to the target
- Meta Data is used in the standard targets, some examples include:
 - Foreground and background colors for the console target
 - Custom headers and subject for the SMTP target

```
var logger = LoggerFactory.GetLogger();

logger.Log("I am default!");

logger.Log("I am black and blue!", new Dictionary<string, object>
{
    { "ConsoleForeground", ConsoleColor.Black },
    { "ConsoleBackground", ConsoleColor.Blue }
});

logger.Log("I am green and yellow!", new Dictionary<string, object>
{
    { ConsoleTarget.MetaForeground, ConsoleColor.DarkGreen },
    { ConsoleTarget.MetaBackground, ConsoleColor.Yellow }
});
```



```
8 : I am default!
8 : I am black and blue!
8 : I am green and yellow!
Done. Press [Enter] to exit
```

Meta Data

- Meta Data is implemented as a set of name/value pairs¹
- The Meta Data is attached to the log event, and therefore is available to the targets which handle the log event
- The base logger provides methods for passing meta data in with the log call

1. IDictionary<string, object>

```

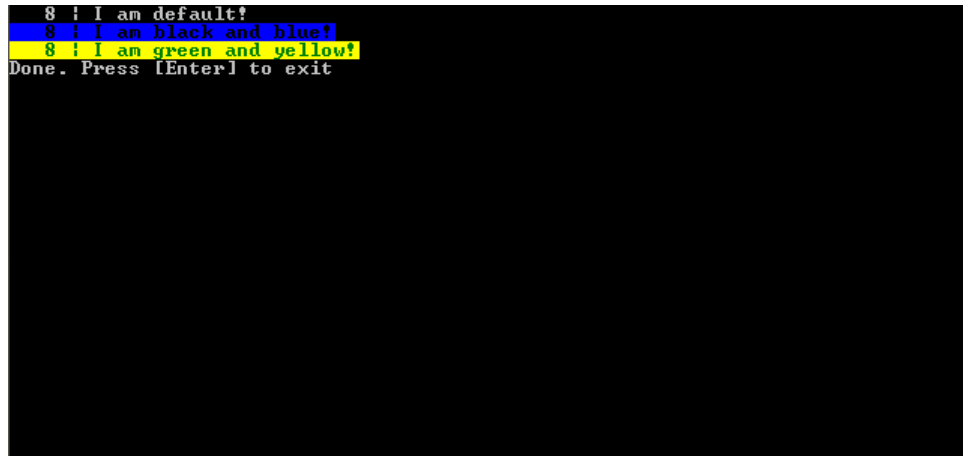
var logger = LoggerFactory.GetLogger();

logger.Log("I am default!");

logger.Log("I am black and blue!", new Dictionary<string, object>
{
    { "ConsoleForeground", ConsoleColor.Black },
    { "ConsoleBackground", ConsoleColor.Blue }
});

logger.Log("I am green and yellow!", new Dictionary<string, object>
{
    { ConsoleTarget.MetaForeground, ConsoleColor.DarkGreen },
    { ConsoleTarget.MetaBackground, ConsoleColor.Yellow }
});

```



```

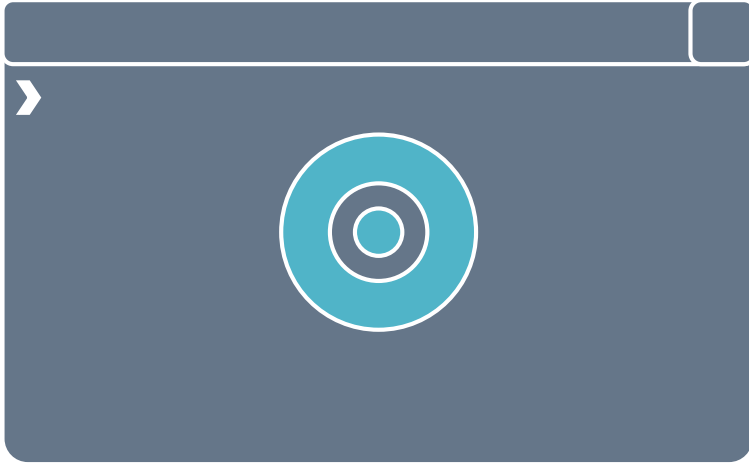
8 : I am default!
8 : I am black and blue!
8 : I am green and yellow!
Done. Press [Enter] to exit

```

Meta Data

- Well-behaved targets will:
 - Function regardless of the Meta Data's presence or format
 - Treat passed-in Meta Data as an override to Meta Data established by Meta Data Providers¹
 - Provide constants for the names/keys of the Meta Data

1. More on Meta Data Providers later in this presentation



“ConsoleForeground”



“ConsoleBackground”

Console Target Meta Data

- The Console Target supports Meta Data:
 - “ConsoleForeground” allows for overriding the foreground color of the console text
 - “ConsoleBackground” allows for overriding the background color of the console text



“EmailSubject”



“EmailHeaders”



“EmailAttachments”



“EmailTo”



“EmailCC”



“EmailBCC”

Email Target Meta Data

- The Email Target supports Meta Data:
 - “EmailSubject” overrides the subject
 - “EmailHeaders” includes additional headers
 - “EmailAttachments” attaches files to the email¹
 - “EmailTo” overrides who the mail is addressed to²
 - “EmailCC” overrides who is copied on the email²
 - “EmailBCC” overrides who is blind copied on the email²

1. A class “EmailAttachment” is provided which allows for defining a byte array, or a file name for attaching

2. A collection of string is expected for these

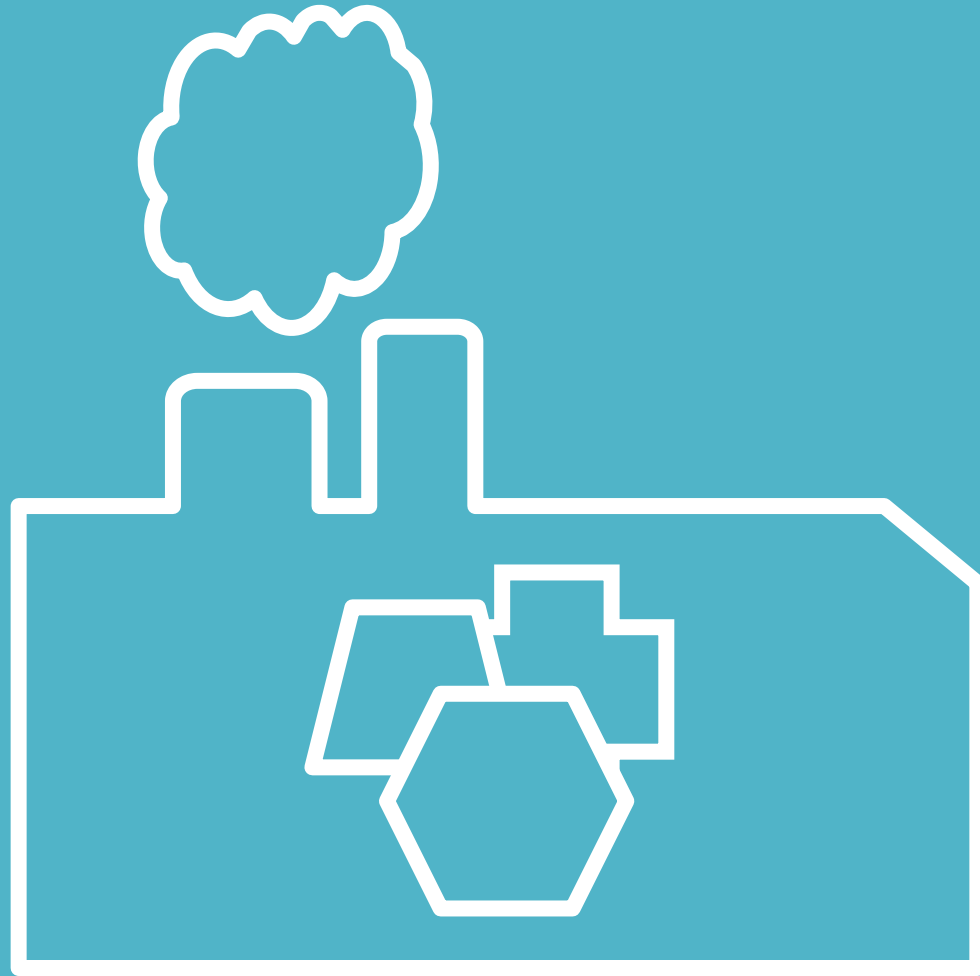


```
var logger = LoggerFactory.GetLogger();  
logger.Log(EmailLogEventBuilder.Create("Hello, World!", "mytag")  
    .AddToAddress("someguy@somewhere.org")  
    .AddCCAddress("someotherguy@somewhere.org")  
    .AddAttachment(new EmailAttachment {  
        PhysicalFileName = "mytextfile.txt",  
        AttachmentFileName = "renamed.txt" })  
    .SetSubject("Hello, Subject!")  
    .SetHeader("someheader", "headervalue")  
    .Build());
```

Email Target Logger Extensions

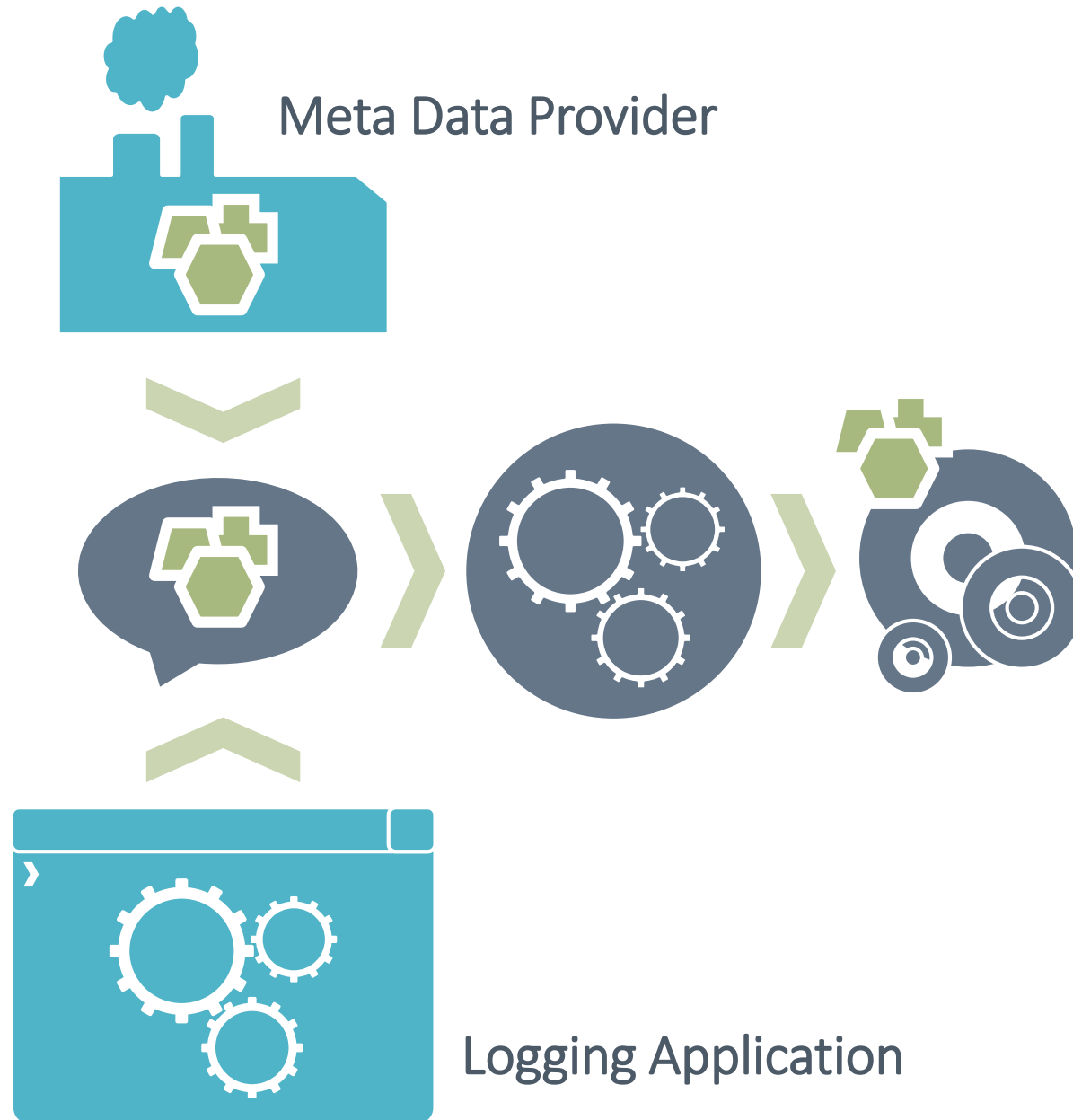
- A helper class is provided that is called EmailLogEventBuilder, which provides for creating a custom log event with the meta data for an email, making sending “custom” emails in code simple¹

1. “using” NuLog.Extensions.Email



Meta Data Providers

If I went to work in a factory the first thing I'd do is join a union



Meta Data Providers

- Meta Data Providers supply meta data for log events
- Meta Data Providers allow for an abstraction of meta data away from the line by line logging, minimizing boilerplate code
- Meta Data Providers are useful for providing additional information to targets such as "Requestor IP" and "Session ID" without the developer having to pass this information with each call to the logger
- Meta Data Providers can be loaded automatically, or manually at runtime¹

1. Meta Data Providers are loaded using the MEF framework in code; or at runtime when requesting the logger from the factory (such as is useful for MVC Controllers)

Home Controller:

```
public class HomeController : Controller, IMetaDataProvider
{
    private LoggerBase Logger
    {
        get
        {
            return LoggerFactory.GetLogger(this); ①
        }
    }

    public HomeController() : base()
    {
        Logger.Log("Controller initialized");
    }

    public ActionResult Index() ③
    {
        Logger.Log("Accessed index");
        return View();
    }

    public ActionResult About() [..]
    public ActionResult Contact() [..]

    public IDictionary<string, object> ProvideMetadata()
    {
        var metaData = new Dictionary<string, object>();

        if (Request != null)
            metaData["RequestIP"] = Request.UserHostAddress;

        if (Session != null)
            metaData["SessionID"] = Session.SessionID; ②

        return metaData;
    }
}
```

JSON Config:

```
"targets": [
  {
    "name": "trace",
    "type": "OID.Common.NuLogging.Targets.TraceTarget",
    "layout": "${DateTime:'{0:MM/dd/yyyy hh:mm:ss.fff}'} | ${RequestIP} | ${SessionID} | ${Message}\r\n"
  },
],
"rules": [
  {
    "include": [ "*" ],
    "writeTo": [ "trace" ],
    "final": false
  }
],
"watch": true ④
```

Trace Target:

```
09/15/2014 07:41:46.987 | | | Controller initialized
09/15/2014 07:41:47.038 | ::1 | tggfhmklex15v2dnovpfvupm | Accessed index
```

Runtime Meta Data Provider

- A practical example of using a runtime Meta Data provider:
 1. The Meta Data provider (the controller itself) is passed into the LoggerFactory when retrieving the logger
 2. The Request IP and Session ID are automatically captured by the controller
 3. The calls to the logger don't have to have to pass the Request IP or the Session ID now
 4. Using a layout target (TraceTarget), the Request IP and Session IP are made available
- All of this is done using the standard components built into the framework, no additional extensions were included or written to achieve this

Home Controller:

```
public class HomeController : Controller, IMetaDataProvider
{
    private LoggerBase Logger
    {
        get
        {
            return LoggerFactory.GetLogger(this);
        }
    }

    public HomeController() : base()
    {
        Logger.Log("Controller initialized");
    }

    public ActionResult Index()
    {
        Logger.Log("Accessed index");
        return View();
    }

    public ActionResult About() [..]

    public ActionResult Contact() [..]

    public IDictionary<string, object> ProvideMetaData()
    {
        var metaData = new Dictionary<string, object>();

        if (Request != null)
            metaData["RequestIP"] = Request.UserHostAddress;

        if (Session != null)
            metaData["SessionID"] = Session.SessionID;

        return metaData;
    }
}
```

JSON Config:

```
"targets": [
  {
    "name": "trace",
    "type": "OID.Common.NuLogging.Targets.TraceTarget",
    "layout": "${DateTime:'{0:MM/dd/yyyy hh:mm:ss.fff}'} | ${RequestIP} | ${SessionID} | ${Message}\r\n"
  },
],
"rules": [
  {
    "include": [ "" ],
    "writeTo": [ "trace" ],
    "final": false
  }
],
"watch": true
```

Trace Target:

```
09/15/2014 07:41:46.987 | | | Controller initialized
09/15/2014 07:41:47.038 | :1 | tggfhmklex15v2dnovpfvupm | Accessed index
```

Runtime Meta Data Provider

- The highlighted sections are simply and easily abstracted out to another class and reused¹:
 - An abstract class, named perhaps `LoggingControllerBase` that implements/extends `Controller` and `IMetaDataProvider`
 - Change the scope of the `Logger` property to protected or higher
 - The concrete controllers would then extend `LoggingControllerBase`, inheriting access to the `Logger` and the populated `Meta Data` for free!

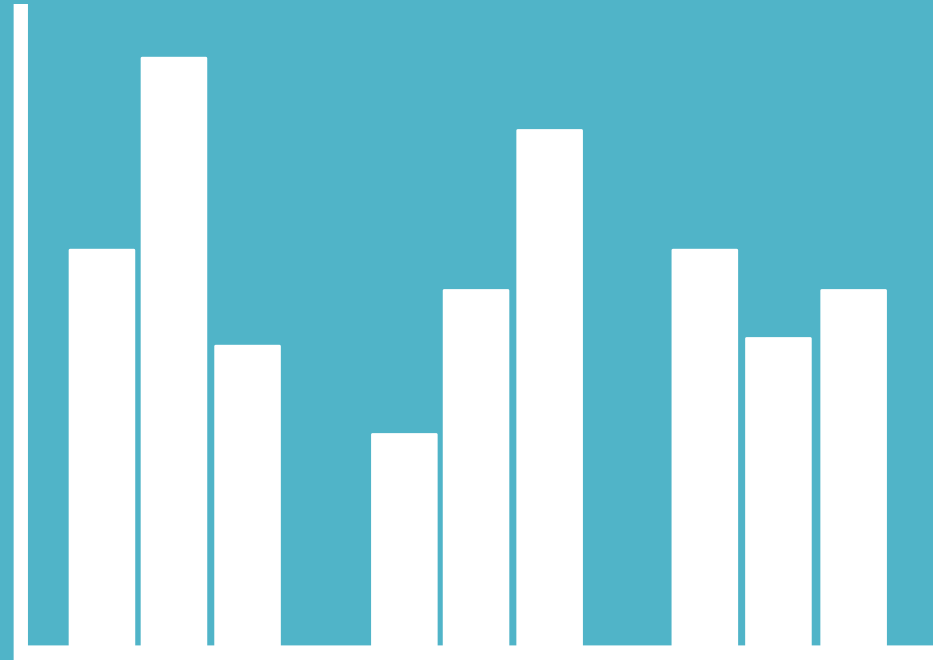
1. This will be done in the simple wrapper, or in the application template

Static Meta Data Provider

```
[Export(typeof(IMetaDataProvider))]  
public class MyStaticMetaData : IMetaDataProvider  
{  
    public IDictionary<string, object> ProvideMetaData()  
    {  
        return new Dictionary<string, object>  
        {  
            { ConsoleTarget.MetaBackground, ConsoleColor.DarkGray },  
            { ConsoleTarget.MetaForeground, ConsoleColor.Green }  
        };  
    }  
}
```

- Static meta data providers are useful for providing meta data at a global level; information that is not request specific, such as machine name, environment or even console color¹
- MEF is used to automatically wire up static meta data providers, no extra wiring (after the “Export” attribute) is required by the implementing developer

1. Practically, this will be helpful for providing information to a DB logger; information like machine name, environment logged-by etc.



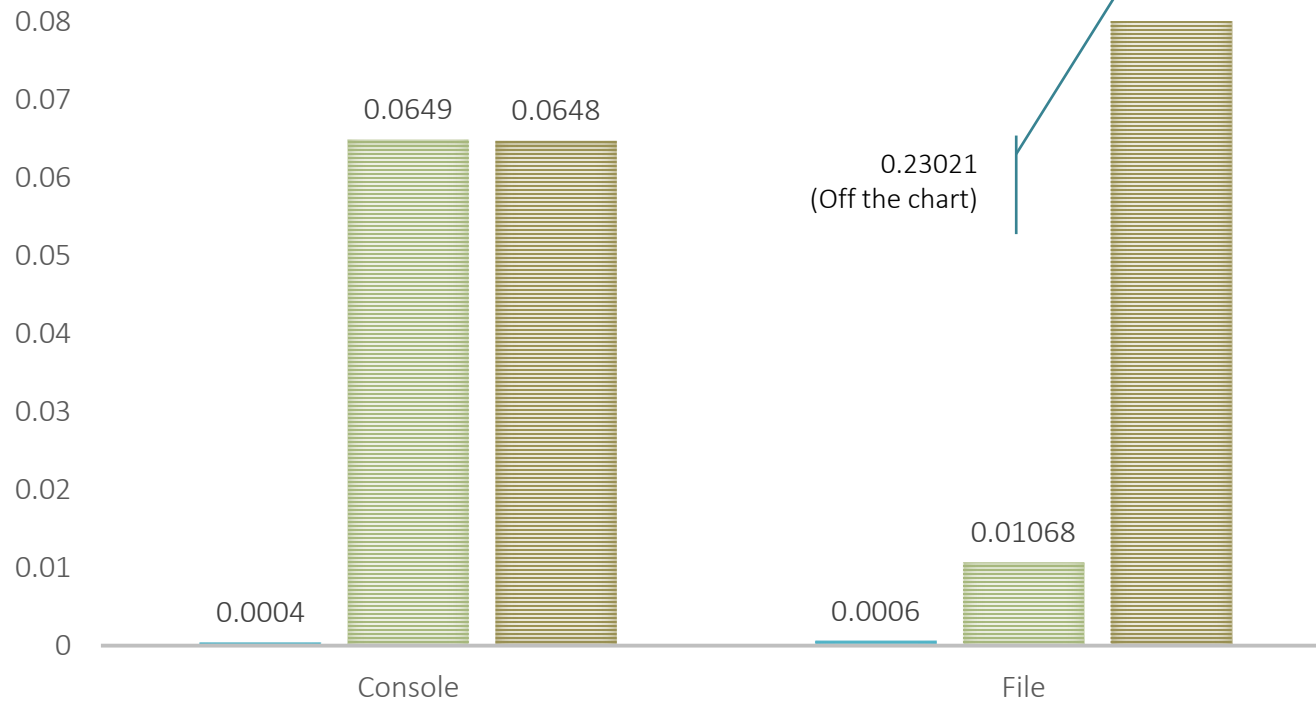
Performance Tests

An ounce of performance is worth pounds of promises

FRAMEWORK COMPARISON

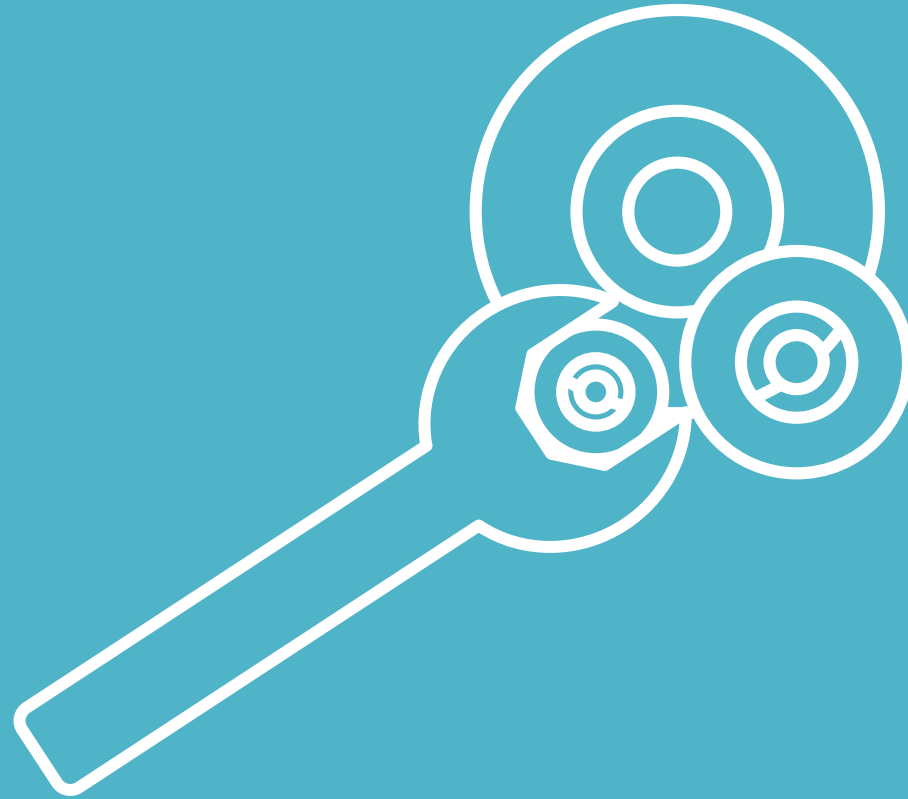
(Milliseconds Per Message Logged)

NuLog Log4Net NLog



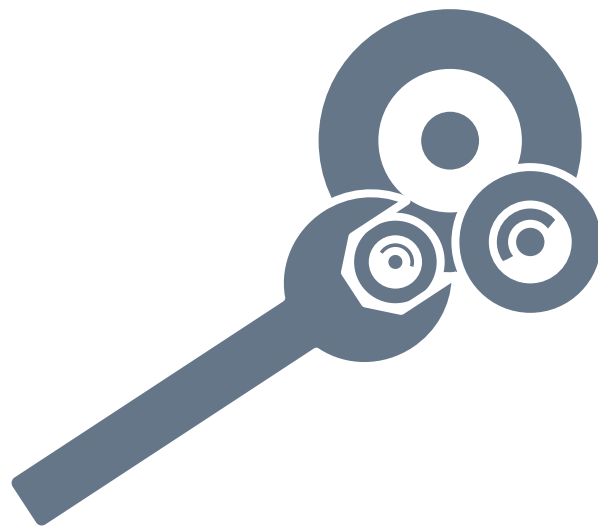
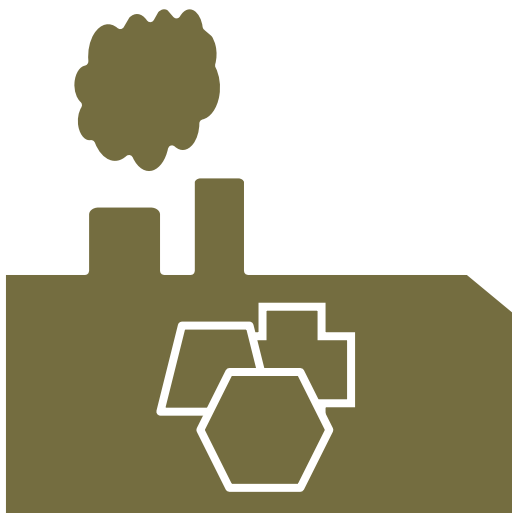
Standard Performance

- When comparing the default asynchronous performance of NuLog to the other major frameworks:
 - NuLog is much faster at handling log events and returning control back to the logging application than the other two
 - NuLog beats Log4Net and NLog in the console by over 6/100 milliseconds per log event
 - NuLog beats Log4Net by over 9/1000 milliseconds per log event, and NLog by a whopping 22/100 milliseconds per log event



Customization and Extension

Like all magnificent things, it's very simple



Customization and Extension

- NuLog is designed from the ground up with extension and customization in mind, and is therefore, very highly customizable and extensible
- The high performance of the framework is inherited by custom targets that are written¹
- Another presentation will be provided which details creating a custom target for the framework which explores much more the details of extending the framework

1. Careful consideration still needs to be made in terms of synchronization when writing custom targets.

Licensing and Credits

It is amazing what you can accomplish if you do not care who gets the credit

NuLog is released under the MIT license (<http://opensource.org/licenses/MIT>) with the understanding that this software depends on other open source libraries that have their own, independent licensing:

Json.NET

MIT License

<http://json.codeplex.com/license>

SharpZipLib

Modified GPL License (Allowing “terms of [our] choice”)

<http://icsharpcode.github.io/SharpZipLib/>

The original author of this library (Ivan Andrew Pointer of Salem Oregon) reserves the right to change this license at any time for any future release of the software and retain full ownership of the software and source code. Any copies you have obtained before any license change will still be governed the license they were released under.

It is not my intent to ever stop providing this library as a free (as in free beer) solution. I may provide enterprise support in the future.

NuLog Licensing

It is amazing what you can accomplish if you do not care who gets the credit
Harry S. Truman

Like all magnificent things, it's very simple
Natalie Babbitt, Tuck Everlasting

Simplicity is the ultimate sophistication
Leonardo da Vinci

Coming together is a beginning; keeping together is progress; working together is success
Henry Ford

Talent hits a target no one else can hit; Genius hits a target no one else can see
Arthur Schopenhauer

Everything should be as simple as it is, but not simpler
Albert Einstein

Do three things well, not ten things badly
David Segrove

Style is a simple way of saying complicated things
Jean Cocteau

The golden rule is that there are no golden rules
George Bernard Shaw

A small group of thoughtful people could change the world. Indeed, it's the only thing that ever has.
Margaret Mead

Growth is never by mere chance; it is the result of forces working together
James Cash Penney

If I went to work in a factory the first thing I'd do is join a union
Franklin D. Roosevelt

An ounce of performance is worth pounds of promises
Mae West

Quotes Attribution

- The quotes shown in this presentation are attributed to the left