Tavisca Logging Framework

Sumit Maingi

This document aims to explain the usage and configuration of the Tavisca Logging Framework.

Tavisca Solutions Pvt Ltd

6/17/2013

Table of Contents

Introduction

The Tavisca logging framework is built to be extensible, fast & aims to consolidate the logging strategy & schema across applications in the Tavisca product suite.

This document will explain the various configurations available in the framework as well as explain the recommended usage strategies.

Configurations

The configuration elements of the logging framework revolve around the interface IApplicationLogSettings, present in the namespace: Tavisca.Frameworks.Logging.Configuration.

Configuration elements are defined below:

Element	Description	Default
ExceptionSwitch	Gets or sets a switch value which determines whether the framework will log exceptions or not. This is a universal switch.	On
EventSwitch	Gets or sets a switch value which determines whether the framework will log events or not. This is a universal switch.	On
ReThrowLogExceptions	Gets or sets a value which determines whether the framework should rethrow the exception once encountered. Setting it to "On" might cause the application to crash if there are too many logging errors, however ideally a sink should not ever throw an exception and should be allowed to crash in case of errors.	On
Categories	Gets or sets the categories defining the loggers along with key value pairs.	-
TraceLoggers	Gets or sets tracing loggers which handles all the trace related logging, the ApplicationTraceListener should be configured.	-
MaxThreads	Gets or sets the maximum number of threads that will be used for logging, if the limit breaches, subsequent requests are queued, this will give a performance boost in most scenarios.	50 (Recommended: 5 for heavy logging.)
MinPriority	Gets or sets the minimum priority required in order for a log to pass through to a sink.	Undefined (lowest, allows all.)
LogConfigurationProvider	Gets or sets an assembly qualified name of a configuration provider; type must implement IConfigurationProvider.	-

	This property can be used to have a custom "settings" provider, when this is provided all other settings are ignored and the custom provider must provide the valid settings.	
CustomLocatorAdapter	Gets or sets an assembly qualified name of a custom service locator, this type must implement IServiceLocator. Available options in the "Extensions" are: 1) CommonServiceLocatorAdapter (required the application to have common service locator usage) 2) SingularityAdapter (for applications using the Tavisca Singularity framework.) 3) ReflectionAdapter (For small applications only, this is NOT recommended for big applications). A custom implementation can inherit Microsoft.Practices.ServiceLocation. ServiceLocatorImplBase and provide its own usage.	- (required)
CustomFormatter	Gets or sets the custom formatter which is called each time by before an entry is logged. The formatter must implement ILogEntryFormatter. For custom formatting also consider inheriting DefaultFormatter and overriding appropriate members. An extension for "CreditCardMaskFormatter" is also present in the extensions assembly. If the masking formatting is required, inherit that class and override the respective members.	DefaultFormatter
DefaultLogger	Gets or sets the default logger for the framework. This comes into play in case the category is passed as "null".	- (optional)
useWorkerProcessThreads	Gets or sets whether the library should use threads from the worker process or create new ones.	- true - (optional) - false to not use wp threads.
compressionType	Defines the compression type to be used. Options are: Zip, Deflate & Custom. If custom is used then "customCompressionType" must be defined. Deflate: uses Encoding.Default for encoding	Deflate

	the binary output. Zip: Uses UTF8 to encode the binary output.	
customCompressionType	Assembly qualified name of the custom compression provider. The name must resolve into a type which implements "Tavisca.Frameworks.Logging.Compression.ICompressionProvider". The instance created will be singleton in nature.	-

The logging framework supports a custom "LogConfigurationProvider" (see the table above), this makes the framework call the custom implementation to get its settings once per application load. Custom code can be written which "refreshes" the settings while the application is running if required, See ILogger.RefreshSettings.

Sample Configurations

Static Application Config mode

```
<configSections>
    <section name="ApplicationLog"
type="Tavisca.Frameworks.Logging.Configuration.ApplicationLogSection,
Tavisca.Frameworks.Logging"/>
  </configSections>

  <ApplicationLog exceptionSwitch="On" eventSwitch="On" maxThreads="5"
                  reThrowLogExceptions="Off" minPriority="Undefined"
    customLocatorAdapter="Tavisca.Frameworks.Logging.Tests.Mock.DummyLocatorAdapter,
Tavisca.Frameworks.Logging.Tests"
    customFormatter="Tavisca.Frameworks.Logging.Tests.Mock.DummyFormatter,
Tavisca.Frameworks.Logging.Tests"
                  defaultLogger="EventViewerLogger">
    <categories>
      <add name="Default">
        <loggers>
          <add name="EventViewerLogger"></add>
        </loggers>
      </add>
      <add name="ServiceLevelLog">
        <loggers>
          <add name="EventViewerLogger"></add>
        </loggers>
      </add>
    </categories>
    <traceLoggers>
      <add name="DBLogger"></add>
    </traceLoggers>
  </ApplicationLog>
```

Custom Provider Sample

```
<configSections>
    <section name="ApplicationLog"
type="Tavisca.Frameworks.Logging.Configuration.ApplicationLogSection,
Tavisca.Frameworks.Logging"/>
  </configSections>

  <ApplicationLog
logConfigurationProvider="Tavisca.Frameworks.Logging.Tests.Custom.TestCustomConfigProv
ider, Tavisca.Frameworks.Logging.Tests.Custom" />
```

Extensions

The extensions project is meant to provide default out of the box classes which are meant to make it easier to get started with the logging framework.

The following extensions are provided in the **Tavisca.Frameworks.Logging.Extensions** assembly.

IOC Containers

The containers are responsible for creating and maintaining the objects that are required by the framework. The following extensions are provided.

CommonServiceLocatorAdapter

The common service locator from the namespace “Microsoft.Practices.ServiceLocation” is commonly used in applications for abstracting the IOC implementation.

The logging extensions provides an implementation which uses the “Current” static property of the “ServiceLocator” object in order to provide instances.

Location: Tavisca.Frameworks.Logging.Extensions.DependencyInjection.Adapters.CommonServiceLocatorAdapter

Pre-Requisites: The service locator needs to be pre-configured, this is meant for applications already using the common service locator or are planning to use the same in their project.

SingularityAdapter

The Tavisca singularity framework is a wrapper around the Unity framework of Microsoft which provides some extended functionalities. The singularity framework is popularly used in the engines in the Tavisca product suite.

The logging extensions provide a default implementation of the same which resolves the type based on the static singularity API.

Location: Tavisca.Frameworks.Logging.Extensions.DependencyInjection.Adapters.SingularityAdapter

Pre-Requisites: The application needs to have been using the singularity framework of Tavisca.

ReflectionAdapter

Meant strictly for tiny to small applications, the reflection adapter uses reflection to create objects based on the key and type.

The reflection adapter requires the logger keys in the config to be “assembly qualified names” of all the “Sink” objects. For the rest the adapter iterates through the types in the current domain and finds the first matching which has a default constructor available. The adapter uses HttpRuntime caching for performance reasons.

The following example shows an example of a typical configuration for a reflection adapter:

```
<ApplicationLog exceptionSwitch="On" eventSwitch="On" maxThreads="5"
                reThrowLogExceptions="Off" minPriority="Undefined"

customLocatorAdapter="Tavisca.Frameworks.Logging.Extensions.DependencyInjection.Adapte
rs.ReflectionAdapter, Tavisca.Frameworks.Logging.Extensions"
    customFormatter="Tavisca.Frameworks.Logging.Tests.Mock.DummyFormatter,
Tavisca.Frameworks.Logging.Tests"
    defaultLogger="Tavisca.Frameworks.Logging.Extensions.Sinks.EventViewerSink,
Tavisca.Frameworks.Logging.Extensions">
    <categories>
      <add name="Default">
        <loggers>
          <add name="Tavisca.Frameworks.Logging.Extensions.Sinks.EventViewerSink,
Tavisca.Frameworks.Logging.Extensions"></add>
        </loggers>
      </add>
    </categories>
    <traceLoggers>
      <add name="Tavisca.Frameworks.Logging.Extensions.Sinks.FileSink,
Tavisca.Frameworks.Logging.Extensions"></add>
    </traceLoggers>
  </ApplicationLog>
```

Location: Tavisca.Frameworks.Logging.Extensions.DependencyInjection.Adapters. ReflectionAdapter

Pre-Requisites: *None.*

Sinks

The sinks are the classes ultimately responsible for pushing the data into a target location. Sinks have to implement the interface “ISink” but it is highly recommended to inherit from “SinkBase” present in the namespace “Tavisca.Frameworks.Logging” and implement its abstract methods.

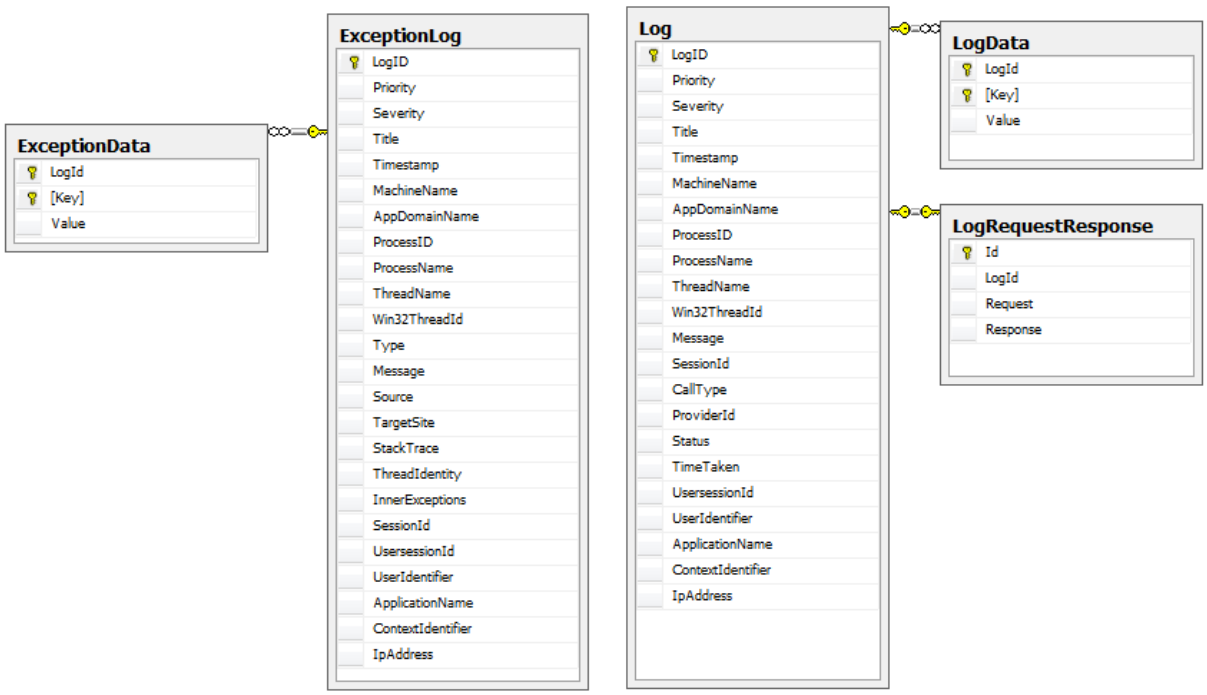
The following Sinks are present in the extensions project for ease of use:

DBSink

The DBSink pushes the data into the database in a predetermined schema.

The schema is given below for your convienience:

(See next page)



Configurations

The DBSink requires a connection string in the web.config the name “log” pointing to a database with the above specified schema.

Example:

```
<connectionStrings>
    <add name="log" connectionString="Data Source=sukhoie;Initial
Catalog=dlogging_ExceptionHandling;Integrated Security=True;MultipleActiveResultSets=True"
providerName="System.Data.SqlClient" />
  </connectionStrings>
```

Notes

The DBSink can be inherited and a function “GetLoggingContext” can be overridden to change the target connection string name in case two databases need to be targeted with different or same category of logging.

SqlSpSink

The SqlSpSink is functionally identical to the DBSink described above. The change lies under the covers where the sink uses stored procedure instead of Entity Framework. This sink is ~10% faster under heavy load, majorly due to lesser CPU consumption caused by the usage of Entity Framework.

Element	Description	Default
Tavisca.Frameworks.Logging.Extensions.Loggers.EventLoggingSP	(optional) The name of the procedure to be used for event logging.	dbo.spWriteLog2
Tavisca.Frameworks.Logging.Extensions.Loggers.ExceptionLoggingSP	(optional) The name of the procedure to be used for exception logging.	dbo.spWriteExceptionLog2

Notes

The SqlSpSink although significantly faster than the DBSink also has certain disadvantages, these are mentioned below for reference:

1) As the name suggests, this implementation is very Sql specific, the DBSink however carries the advantage of Entity Framework by being cross DB compatible.
2) The SP sink also loses out on the very verbose data validation features of the DBSink, wherein data would be validated against the DB model, e.g. if a field is more than the allowed limit, the SP logger would inherently truncate the text and push it, in case of DB logger however, an exception would be raised which would provide details of the validation failure.

SqlSpBufferredSink

Inherits from SqlSpSink adds the functionality of buffering requests for 2 seconds and pushing the log entries every 2 seconds under a single opened connection for performance benefits.

Notes

1) All the settings for this sink remain the same of that of SqlSpSink.
2) This sink is a sealed class and cannot be extended.

RedisSink

Pushes items into a Redis list after serializing the data into JSON. This sink can be used with the ELK stack (or anything else) for performance benefits. The serialized output is further compressed using "fast" mode of gzip compression.

Element	Description	Default
`Logging.RedisServerConnString`	(Required) The Redis connection string*	-
`Logging.RedislistKey`	(Required) The name of the Redis list in which the logs will be pushed.	-

Redis Connection Strings

Redis connection string is meant to define the server address in which the logs will be dumped.

Standard format:

<ip address>:<port number> e.g. 192.168.0.0:1234

The port number is optional and defaults to "6379" if not provided. If port number is not to be given, the connection string can be defined as "*192.168.0.0*".

Multiple addresses of the above format can be provided separated by ",".

<ip address>:<port number>,<ip address2>:<port number2>,….

In the above scenario logs will be evenly distributed among the provided servers.

URL Format:

<http address>:<port number>,<http address2>:<port number2>

All the rules apply for the previous formats, in addition to the latter the http addresses are DNS resolved and kept cached until a connection error occurs, the DNS resolution must be successful and give a valid IP address. This can be used in conjunction with Route 53 service for AWS or similar services for Cloud management ease.

Machine Name Format:

<machine name>:<port number>,<machine name2>:<port number2>,….

All the above rules apply, behaves exactly the same way as http format given above.

Multiple IP Address Resolution

In the above examples whenever the DNS service (think AWS Route 53) resolves multiple IP addresses, all these addresses are then treated as multiple connections i.e. with every push the sink will choose a random connection to push into; note that the port number in this case remains the same for the set, if port numbers are to be changed, comma separated feature as described before must be used.

DNS resolution in the framework is cached for 5 minutes, also note that there is caching in a DNS service as well, typically that of 5 minutes, the former two statements together mean that in the worst case the effect of a modification in the DNS service can take as much as 10 minutes.

The following additional assemblies need to be referenced in the host project(s) when using this sink:

1) Newtonsoft.Json
2) ServiceStack.Common
3) ServiceStack.Interfaces
4) ServiceStack.Redis
5) ServiceStack.Text
6) Tavisca.Frameworks.Helper

EventViewerSink

The event viewer sink logs the data as windows events which are typically viewed in the “Event Viewer” (Run console + type “eventvwr” + <Enter>). The data can be seen in the “Application” section.

Configurations

The event viewer sink has an optional configuration which allows the user to specify the name of the “source” of logging. The configuration should be present in the “**appsettings**” section of the application configuration file.

Element	Description	Default
`Tavisca.Frameworks.Logging.Extensions.Loggers.EventViewerLogger.Source`	(optional) The name of the source as shown in the event viewer.	Tavisca.Frameworks.Logging.Extensions.

Notes

The class can be inherited and implementation of the translation of the "entry" objects into text can be changed by returning a custom implementation of "Tavisca.Frameworks.Logging.Extensions.Infrastructure.IEntryStringTranslator" in the function "GetTranslator". The default implementation uses the "IOC adapter" to create the object.

Optionally change the default implementation of "IEntryStringTranslator" in the IOC container to a custom implementation.

Also both the "FileSink" as well as the "EventViewerSink" inherits from "StringWritingSinkBase" in the same namespace which provides the translation for converting entity objects into their string representations.

FileSink

The file sink pushes the data into a file. The sink provides "rollover" functionality on a daily basis.

Configuration

The configurations are present in the "**appsettings**" in the application configuration file.

Element	Description	Default
`Tavisca.Frameworks.Logging.Extensions.Loggers.FileLogger.FilePath`	(Required) Defines the file path in which the logs will be written. The process must have access to edit & create files in that location.	-
`Tavisca.Frameworks.Logging.Extensions.Loggers.FileLogger.MaxFileSize`	(Optional) Defines the maximum size of the file on a daily basis. After the limit breaches, no further logs are written for the rest of the day until the day goes over and a new file is created for the day (rollover). This is for disk management. The size is in "bytes".	`10485760 (10 Mb)`

Notes

The class can be inherited and implementation of the translation of the "entry" objects into text can be changed by returning a custom implementation of "Tavisca.Frameworks.Logging.Extensions.Infrastructure.IEntryStringTranslator" in the function "GetTranslator". The default implementation uses the "IOC adapter" to create the object.

Optionally change the default implementation of "IEntryStringTranslator" in the IOC container to a custom implementation.

Also both the "FileSink" as well as the "EventViewerSink" inherits from "StringWritingSinkBase" in the same namespace which provides the translation for converting entity objects into their string representations.

Formatters

Formatters are called before any entry goes into a sink configured in the system, the formatter's serves as an injection point to add/ edit data in an "entry" object before it is logged. The "**DefaultFormatter**" provided in the core logging framework does two jobs:

1) It is responsible for compressing the request and response of the "IEventEntry" object.
2) It is responsible for converting an exception into an "IExceptionEntry" object.

The default formatter functionality is deemed basic and all custom formatters should inherit from the same and only extend its functionality. The way to preserve the default functionality is by overriding its methods and calling the "base" method at the end return statement.

The extension project formatter's builds upon the above mentioned formatter and chains further functionality into it.

Currently there is only one formatter provided by the extension project:

CreditCardMaskFormatter

The CreditCardMaskFormatter inherits from the "DefaultFormatter" (see above) and *extends* its functionality to provide masking functionality in the request and response fields of the "IEventEntry" object.

The default out-of-the-box behaviour of the formatter is to look at each and every response and mask the "credit card numbers" to show only the last four digits.

The formatter also supports a provider based model wherein a custom implementation of "Tavisca.Frameworks.Logging.Extensions.Formatters.ICreditCardMaskDataProvider" can be provided (see the configuration section). A helper base class is present at "Tavisca.Frameworks.Logging.Extensions.Formatters.CreditCardMaskDataProviderBase" for ease of implementation.

The formatter class takes care of caching for the formatter data. The caching is for hardcoded duration of 1 hour.

Configuration

The configuration is in the "**appsettings**" section of the application configuration file.

Element	Description	Default
`Tavisca.Frameworks.Logging.Extensions.Formatters.ICreditCardMaskDataProvider`	(optional) The name of the provider for the masking functionality.	-

Usage

Usage of the logging framework is designed to be as simple as possible while keeping the flexibility of the system aligned to its goals.

See the configuration section for a sample configuration.

IOC Configurations

The following elements need to be configured in the IOC container (unless reflection adapter is being used {only for tiny applications} which requires none of these).

Required Implementation	Default Implementation	Notes
Tavisca.Frameworks.Logging.Extensions.Infrastructure. IEntryStringTranslator	Tavisca.Frameworks.Logging.Extensions.Infrastructure.EntryStringTranslator	Required if either of FileSink or EventViewerSink is being used from the Extensions project.
Tavisca.Frameworks.Logging.IExceptionEnt	Tavisca.Frameworks.Logging.ExceptionEntry	Required.

ry		
Tavisca.Frameworks.Logging.IEventEntry	Tavisca.Frameworks.Logging.EventEntry	Required.
Tavisca.Frameworks.Logging.Tracing.ITraceLogger	Tavisca.Frameworks.Logging.Tracing.TraceLogger	Required if the framework is also being utilized for application tracing.
Tavisca.Frameworks.Logging.ISink	DBSink EventViewerSink FileSink SqlSpSink	Required, the sinks will be called by the given "name" or key as given in the configuration.

Code Sample

The samples given here give a hint to the recommended usage of the logging framework.

The usage:

```
static void Main(string[] args)
        {
            IEventEntry entry = null;
            try
            {
                entry = Utility.GetLogEntry();

                entry.Title = "Making a sample of the logging framework.";

                entry.CallType = "Entering some call type.";

                entry.ProviderId = 10; //entered some provider Id.


                //do stuff

                entry.TimeTaken = 10; //entered a time taken
            }
            catch(Exception ex)
            {
                Utility.GetLogger().WriteAsync(ex.ToContextualEntry(), null); //null
category causes the framework to pick up the default logger.
            }
            finally
            {
                Utility.GetLogger().WriteAsync(entry,
KeyStore.Categories.ServiceLevel); //specify a category
            }
        }
```

Important points to note here

1) The logger instance is fetched from a central code, in this example a static utility class.
    a. The central code ensures that a custom logger, inheriting either from the framework provided "Logger" class (recommended) or a more specific implementation of the ILogger interface (not recommended).
    b. The central implementation also ensures that additional functionalities of the logger can be utilized later with a simple change. E.g. setting the "OverriddeEntryFilters" Boolean property of the logger in a specific scenario (and returning that object in

that scenario only) would override all filters that may have been provided into the framework and hence detailed logging in specific scenarios is made possible.

2) The “entry” objects are fetched from a central place, this ensures certain “contextual” properties to be pre-set, this would be more obvious from the below sample of a utility class used in the code above.

```
public static class Utility
    {
        private static readonly ILogger DefaultLogger = new Logger();

        private static readonly ILogger DoAllLogger = new Logger(true);

        public static ILogger GetLogger()
        {
            var var = Environment.GetEnvironmentVariable("someVariable");

            if (string.IsNullOrWhiteSpace(var))
                return DefaultLogger;

            return DoAllLogger;
        }

        public static IExceptionEntry ToContextualEntry(this Exception exception)
        {
            var entry = exception.ToEntry();

            FillEntry(entry);

            return entry;
        }

        public static IEventEntry GetLogEntry()
        {
            var entry = new EventEntry();

            FillEntry(entry);

            return entry;
        }


        public static void FillEntry(ILogEntry entry)
        {
            entry.PriorityType = PriorityOptions.Medium;

            entry.IpAddress = string.Empty;

            entry.UserIdentifier = string.Empty;

            entry.UserSessionId = string.Empty;
        }
}
```