QLog

Quick logging for Windows Azure .NET applications

Version: 1.4.1.0

Documentation

Author: Adam Sobaniec (sobanieca@gmail.com)

Table of Contents

1 Introduction

Logging events in an application is a must if you are developing a complex system. With feedback sent from your source code, it's way easier to understand how it behaves and works. Every unexpected path in an application execution should have a corresponding message in a log to make it possible to track what went wrong when user tried to execute specific part of your code. The more complex your application becomes, the more types of logs it should collect. It's a good practice to separate events into 6 main areas: **TRACE**, **DEBUG, INFO, WARN, ERROR** and **CRITICAL**. **TRACE** area represents the lowest level of logging. It's convenient to store here information about methods which were called and the parameters that were passed to them. **DEBUG** represents all events that collect debug specific data, like how many particular objects where fetched from the database, or what was the response received from a remote server. **INFO** events, are events that inform about standard user actions, for example, to notify when user uploads content to the application. **WARN** events correspond to the unexpected behavior of an user, like when he wants to update an item that doesn't belong to him. **ERROR** category corresponds mainly to exceptions that are being thrown when an application runs. **CRITICAL** area holds events that are critical for the application, if such events occurred then it means that application cannot work correctly and it requires immediate reaction.

1.1 About QLog

QLog is designed to be a very **simple to use and flexible** logging framework for any Windows Azure .NET application. It is not supposed to be a competition for more advanced logging solutions like log4net or nLog which offer many kinds of loggers (file loggers etc.). QLog's aim is to allow developers to log all events in a simple manner by minimizing the required XML configuration and letting your applications *.config files stay clean and readable. One may say that simplicity can lead to the problems when application scales to a very large size. In such case there may be a circumstances when particular logging framework slows down an application because there are a lot of operations executed by it. QLog is designed to **avoid such problems**. It is flexible enough to allow developers to set it up in such way that it will be able to handle big load without significant decrease in performance.

1.2 Requirements

QLog requires .NET Framework 4.0 or newer.

All logs are stored in Azure Storage Tables, so you will need storage account. It is recommended that you create a storage account only for QLog, separating logs from your application main data.

2 Logging with QLog

Logging with QLog is supposed to be as simple as adding reference to the QLog.dll. It is recommended that you do this through NuGET package manager. After doing it, it's enough to add two configuration keys in order to start:

Connection string:

```
<add name="QLogDataSource" connectionString="-connection string-"/>
```

Application setting:

```
<add key="QLogArea" value="QTrace"/>
```

After adding proper keys to the application configuration file, QLog will be ready to log all events from all areas starting from the QTrace (which is the area with lowest severity). You will read more about areas of logging and their severity later.

Now anywhere in your code, if you want to log particular kind of event you can place the following sample line of code:

```
QLog.Logger.LogTrace("/User/Messages controller method called");
```

This instruction tells logger to log an event at the trace area with a specified message. QLog allows to log events on 6 basic areas (the ones mentioned in the introduction) and to log events using custom areas, hence there are following basic commands available:

```
Qlog.Logger.LogTrace("{Message that you want to log}");
QLog.Logger.LogDebug("{Message that you want to log}");
QLog.Logger.LogInfo("{Message that you want to log}");
QLog.Logger.LogWarn("{Message that you want to log}");
QLog.Logger.LogError("{Message that you want to log}");
QLog.Logger.LogCritical("{Message that you want to log}");
```

Each instruction has the following overloaded definition:

```
public void LogYYYYYY(string message);
public void LogYYYYYY(string message, params object[] args);
public void LogYYYYYY(Exception e);
public void LogYYYYYY(string message, Exception e);
```

The second one, applies the popular String.Format() method in order to inject the specified parameters inside the message. The sample usage might look as following:

```
public List<UserMessage> GetUserMessages(User _user)
{
  QLog.Logger.LogTrace("GetUserMessages() called. User id provided: {0}", _user.Id);

  try
  {
    List<UserMessage> userMessages = _messageRepository.GetUserMessages(_user);
    QLog.Logger.LogDebug("{0} user messages received", userMessages.Count);
    return userMessages;
  }
  catch (Exception e)
  {
    QLog.Logger.LogError("Exception thrown when trying to load user messages.", e);
    return null;
  }
}
```

Each log entry contains following fields:

```
public class QLogEntry
{
    public long Id { get; set; }
    public string Message { get; set; }
    public string Area { get; set; }
    public string AreaColor { get; set; }
    public string ThreadId { get; set; }
    public string SessionId { get; set; }
    public string UserAgent { get; set; }
    public string UserHost { get; set; }
    public string Class { get; set; }
    public string Method { get; set; }
    public string InstanceId { get; set; }
    public string DeploymentId { get; set; }
    public DateTime CreatedOn { get; set; }
}
```

As you can see, there is being stored some additional information. First of all Area and AreaColor are the fields that represent the current area of log. The values here can be: „QTrace”, „QDebug”, „QInfo”, „QWarn”, „QError”, „QCritical” or the custom area name if you have created one (you can read more about custom areas in the chapter “Standard and custom logging areas”). AreaColor is an RGB color definition that is used for display in the QlogBrowser. ThreadId is the current managed thread id that called the LogYYYYY() method. SessionId, UserAgent and UserHost are additional values that describe the HTTP request that is related with the current LogYYYYY() call. However, in case when Qlog.Desktop is used, those values are related to the current Windows user. You will read more about exact values stored here in chapter “QLog components”. Class and Method fields represent the class and the method of an object that has called the LogYYYYY(). They are both read from the stacktrace. In case when anonymous type called the LogYYYYY() method, these fields will be empty. InstanceId and DeploymentId are Windows Azure instance and deployment related information. Finally there is a field CreatedOn that represents the exact date when an event occurred.

REMARK

All dates in QLog are stored in UTC format. In QLogBrowser all dates are transformed to the current user local time.

3 QLog components

In order to allow logging events, QLog uses following components:

IConfig
IBuffer
IEnvironment
IRepository

IConfig component is responsible for reading the main configuration for a logger. IBuffer is a component that handles in-memory buffer operations like storing logs (if asynchronous log writing was enabled, refer to the “Configuration” chapter for more information). IEnvironment component is responsible for creating the Log entries that will be saved to the database. It is responsible for setting correct values to the SessionId, UserAgent and other fields. IRepository component is a component that provides an interface for saving all log entries directly to the database. It provides also an interface for additional QLog utilities.

4 Standard and custom logging areas

One of the most significant features of the QLog is the possibility to log events within the custom areas. For example if you are developing a complex system that is built not only from MVC application but also implements an XML and JSON API and you want to log all requests and responses for each API version you can place such events in the following areas:

QJsonApiRequests

QXmlApiRequests

In order to log events within these areas, you need to add the following classes to your code:

```
public sealed class QJsonApiRequests : QArea
{
    public override QAreaColor AreaColor {get { return new QAreaColor(118, 162, 0); }}
    public override uint Severity {get { return 500; }}
}

public sealed class QXmlApiRequests : QArea
{
    public override QAreaColor AreaColor {get { return new QAreaColor(10, 72, 52); }}
    public override uint Severity {get { return 500; }}
}
```

As you can see each custom area can have the default color set (it is used by the QLog Browser) and the severity. The severity of a default areas looks as follows:

QTrace, severity = 0

QDebug, severity = 1000

QInfo, severity = 3000

QWarn, severity = 4000

QError, severity = 5000

QCritical, severity = 6000

Setting up a severity for custom areas is important if you are setting the “QLogArea” configuration value to the single particular area. In such case all events that has lower severity won't be logged. For example If you set the “QLogArea” key to the “QInfo” value, then all events with severity lower than 3000 won't be logged, so the events from “QDebug” and “QTrace” will be ignored. By default all custom areas have their Severity set to 2000. In case If you need to change the severity of a particular area, you have to do it in the same way as in the sample area definitions above.

REMARK

It is recommended to name all custom areas with the “Q” prefix. By doing it, you will avoid the confusion with the names of the classes that are actually performing particular role in your application. For example “QDebug” will not be confused with “Debug” class that is already defined in the System.Diagnostics namespace.

After defining custom areas all that you have to do in order to log event in selected area is to use the following syntax:

```
QLog.Logger.Log<QJsonApiRequests>("Request: {0}.\n\nResponse: {1}", request, response);
```

As you can see QLog logger definition contains the following method:

```
public void Log<Area>(string msg) where Area : QArea
```

The area generic parameter is supposed to inherit from a base class QArea. Method Log<Area>() is overloaded in the same way as the LogYYYYY() methods described earlier. In fact all methods like LogTrace(), LogDebug(), ... are an alias for a methods Log<QTrace>(), Log<QDebug>() etc.

As you can see custom areas allows QLog to be very flexible and easily adapt to the specific needs for a current project (especially if it is n-tier application). They are especially handy when you are dealing with a complex system and you want to keep all ypes of logs in one database. Having all logs in a single place simplifies the whole bunch of administration processes (backups, cleanup etc.), and allows you to browse logs more easily. QLog Browser, attached in the QLog package, is designed to handle custom areas in a simple and readable manner.

5 Configuration

5.1 Connection string

In order to work, QLog requires few basic settings. First of all it needs a correct connection string that points to the database that is going to be used to store all logs:

```
<add name="QLogDataSource" connectionString="" />
```

“QLogDataSource” needs to be declared in web.config file of your application. Place it in the <connectionStrings> section of your application configuration.

5.2 Log areas

After setting connection string, you need to specify the areas of logs for which you want to perform logging:

```
<add key="QLogArea" value="QTrace"/>
```

Defining areas can be done in many ways. First of all you can define a single area (from the set of default areas), like: “QTrace”, “QDebug”, “QInfo”, “QWarn”, “QError” or “QCritical”. If you provide such value, logging will be done basing on the severity of a given area. If you provide the “QInfo” area then there will be no logs for areas “QTrace” or “QDebug” collected because they have lower severity (for a severity values for each area refer to the previous chapter - “Standard and custom logging areas”).

You can provide also two special values: “None” (this values is set by default if no “QLogArea” setting is present) and “All”. If “None” is set, then logger will be disabled. In case when “All” is provided, all logs from all areas will be taken into account.

There may be also a need to explicitly list the areas for which you want to log events. You can do it in two different ways:

“Accept:QWarn, QError, QCritical, QCustomAreaName”

or

“Ignore:QTrace, QDebug”

As you can see you can specify the areas names separated by the comma. If you provide “Accept:” at the beginning, then only areas mentioned in the value will be taken into account. In case when you provide “Ignore:” at the beginning, then the areas provided in the value will be omitted during logging.

5.3 Asynchronous logging

Another QLog setting that can be provided is the asynchronous logging. If you provide the following setting:

```
<add key="QLogAsync" value="True"/>
```

Then all logs from all areas will be saved asynchronously. This means that they will be first saved to the in-memory buffer and later flushed to the database. In order to flush logs to the database, Flush() method must be called. This is being done automatically if HttpApplication (or MvcApplication) inherits from the provided QLogHttpApplication. Otherwise you need to remember about calling the Flush() method manually.

If you set "QLogAsync" value to "False" then no logs will be logged asynchronously it means that each time when Logger.LogYYYY() method will be called, there will be done instant connection to data source which may slow down some algorithms.

You can also provide the list of areas that you want to log asynchronously. The syntax is very similar to the one for "QLogArea", namely:

"Accept:QTrace, QDebug, QInfo, QCustomAreaName"

or

"Ignore:QError, QCritical"

You can specify here explicitly which areas should or shouldn't be logged asynchronously.

5.4 Silent mode

By default QLog has silent mode enabled which means that the only exception that will be thrown by it, will be a QLogNotInitializedException which tells that QLog components were not loaded correctly, or QLogSilentModeHandleException which tells that QLog was unable to determine if silent mode is enabled or disabled. No other exceptions will be thrown. In case when you want to know why your logs are not saved to the database you can turn the silent mode off with the following setting:

```
<add key="QLogSilentMode" value="False"/>
```

Now silent mode will be disabled and you will be able to find out the exceptions that are being thrown by logger.

Always remember to turn this setting back to "True" when deploying to the production environment. It will help you to avoid the situations when your application stops working because, for example, logs data source is inaccessible.

5.5 Stacktrace usage

As mentioned earlier, QLog uses stacktrace to find out which object called the Log() method. In extreme situations this might be a performance hit, so you can disable it by providing the setting:

```
    <add key="QLogStacktrace" value="False"/>
```

In such case, fields Class and Method of each log entry will be empty, but you will get the performance boost.

5.6 QLog postfix

You can define a postfix that will be appended to the table name in case when you want to separate logs from different roles (worker roles for instance), but use the same storage account. By default table name is “QLog”, but you can change it with following configuration setting:

```
<add key="QLogPostfix" value="-postfix for table-"/>
```

5.7 In-memory buffer max count

Each time when asynchronous logging is being done, QLog checks if the number of elements in the in-memory buffer doesn't exceed the specified limit. By default the limit is set to 100, but it can be modified with the following setting:

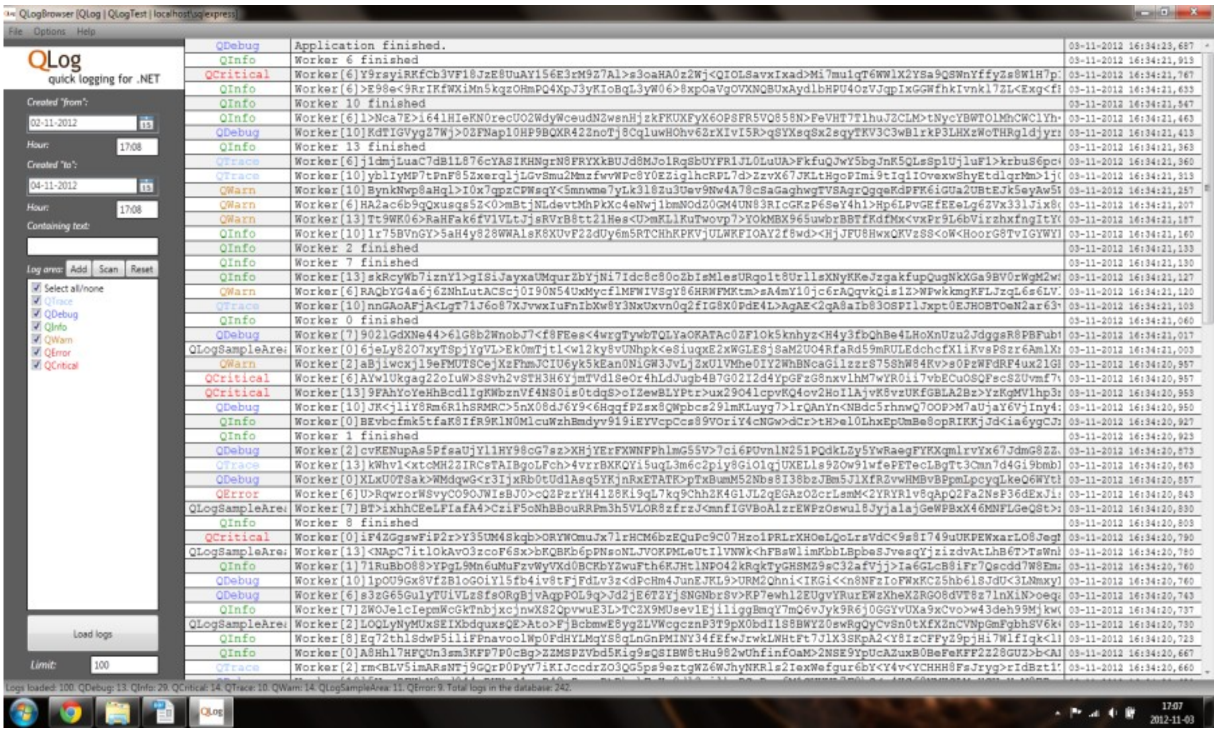
```
<add key="QLogBufferMaxCount" value="-Any Int32 valid value-"/>
```

If the buffer is full, then QLog automatically calls the Flush() method in order to free some space in the buffer. This mechanism was implemented for safety purpose - in case when programmer forgot about placing the Flush() method in a proper place. Max buffer setting can only be set in web.config or app.config file. One important notice: it is not being read from the Azure service configuration.

You can learn about all configuration settings from the Configuration.pdf file that is attached to the QLog.zip package.

7 Browsing logs with QLogBrowser

Inside QLog package you can find an application QLog Browser that allows you to browse all logs collected by your application in a very simple and readable manner.



Created "from": 02-11-2012
Hour: 17:08
Created "to": 04-11-2012
Hour: 17:08
Containing text:
Log area:
☑ Select all/none
☑ QTrace
☑ QDebug
☑ QInfo
☑ QWarn
☑ QError
☑ QCritical
Limit: 100

```
QDebug          Application finished.                                   03-11-2012 16:34:23,687
QInfo           Worker 6 finished                                       03-11-2012 16:34:21,913
QCritical       Worker[6]Y9rsyiRKfCb3VF18JzE8UuAY156E3rM9Z7Al>s3        03-11-2012 16:34:21,767
QInfo           Worker[6]>E98e<9RrIKfWXiMn5kqzOHmPQ4XpJ3yKIoBqL3        03-11-2012 16:34:21,633
QInfo           Worker 10 finished                                      03-11-2012 16:34:21,547
QInfo           Worker[6]l>Nca7E>i641HIeKN0recUO2WdyWceudNZwsnHj        03-11-2012 16:34:21,463
QDebug          Worker[10]KdTIGVyg27Wj>0ZFNap10HP9BQXR42ZnoTj8C         03-11-2012 16:34:21,413
QInfo           Worker 13 finished                                      03-11-2012 16:34:21,363
QTrace          Worker[6]j1dmjLuaC7dB1L876cYASIKHNgrN8FRYXkBUJd8        03-11-2012 16:34:21,360
QTrace          Worker[10]yblIyMP7tPnF85ZxerqljLGvSmu2MmzfwvWPc8        03-11-2012 16:34:21,313
QWarn           Worker[10]BynkNwp8aHql>I0x7qpzCPWsqY<5mnwme7yLk         03-11-2012 16:34:21,257
QWarn           Worker[6]HA2ac6b9qQxusqs5Z<0>mBtjNLdevtMhPkXc4e         03-11-2012 16:34:21,207
QWarn           Worker[13]Tt9WK06>RaHFak6fV1VLtJjsRVrB8tt21Hes<         03-11-2012 16:34:21,187
QInfo           Worker[10]1r75BVnGY>5aH4y828WWAlsK8XUvF2ZdUy6m5         03-11-2012 16:34:21,160
QInfo           Worker 2 finished                                       03-11-2012 16:34:21,133
QInfo           Worker 7 finished                                       03-11-2012 16:34:21,130
QInfo           Worker[13]skRcyWb7iznY1>gISiJayxaUMqurZbYjNi7Id         03-11-2012 16:34:21,127
QWarn           Worker[6]RAQbYG4a6j6ZNhLutACScj0I90N54UxMycflMF         03-11-2012 16:34:21,120
QTrace          Worker[10]nnGAoAFjA<LgT71J6o87XJvwxIuFnIbXw8Y3N         03-11-2012 16:34:21,103
QInfo           Worker 0 finished                                       03-11-2012 16:34:21,060
QDebug          Worker[7]9021GdXNe44>61G8b2WnobJ7<f8FEes<4wrgTy         03-11-2012 16:34:21,017
QLogSampleAre:  Worker[0]6jeLy8207xyTSpjYgVL>Ek0mTjt1<w12ky8vUN         03-11-2012 16:34:21,003
QWarn           Worker[2]aBjiwcxj19eFMUTSCejXzFhmJCIU6yk5kEan0N         03-11-2012 16:34:20,957
QCritical       Worker[6]AYwlUkgag22oIuW>SSvh2vSTH3H6YjmTVd1SeO         03-11-2012 16:34:20,957
QCritical       Worker[13]9FAhYoYeHhBcdlIgKWbznVf4NS0is0tdqS>oI         03-11-2012 16:34:20,953
QDebug          Worker[10]JK<jliY8Rm6R1hSRMRC>5nX08dJ6Y9<6HqgfP         03-11-2012 16:34:20,950
QInfo           Worker[0]BEvbcfmk5tfaK8IfR9KlN0MlcuWzhBmdyv919i         03-11-2012 16:34:20,927
QInfo           Worker 1 finished                                       03-11-2012 16:34:20,923
QDebug          Worker[2]cvKENupAs5PfsaUjYl1HY98cG7sz>XHjYErFXW         03-11-2012 16:34:20,873
QTrace          Worker[13]kWhvl<xtcMH2ZIRCsTAIBgoLFch>4vrrBXKQY         03-11-2012 16:34:20,863
QDebug          Worker[0]XLxU0TSak>WMdqwG<r3IjxRb0tUdlAsq5YKjnR         03-11-2012 16:34:20,857
QError          Worker[6]U>RqwrorWSvyCO9OJWIsBJ0>cQZPzrYH4128Ki         03-11-2012 16:34:20,843
QLogSampleAre:  Worker[7]BT>ixhhCEeLFIafA4>CziF5oNhBBouRRPm3h5V         03-11-2012 16:34:20,830
QInfo           Worker 8 finished                                       03-11-2012 16:34:20,803
QCritical       Worker[0]iF4ZGgswFiP2r>Y35UM4Skqb>ORYWOmuJx71rH         03-11-2012 16:34:20,790
QLogSampleAre:  Worker[13]<NApC7itlOkAvO3zcoF6Sx>bKQBKb6pPNsoNL         03-11-2012 16:34:20,780
QInfo           Worker[1]71RuBbO88>YPgL9Mn6uMuFzvWyVXd0BCKbYZwu         03-11-2012 16:34:20,760
QDebug          Worker[10]1pOU9Gx8VfZB1oGOiY15fb4iv8tFjFdLv3z<d         03-11-2012 16:34:20,760
QDebug          Worker[6]s3zG65GulyTUiVLzSfsORgBjvAqpPOL9q>Jd2j         03-11-2012 16:34:20,743
QInfo           Worker[7]ZWOJelcIepmWcGkTnbjxcjnwXS2QpvwuE3L>TC         03-11-2012 16:34:20,737
QLogSampleAre:  Worker[2]LOQLyNyMUxSEIXbdquxsQE>Ato>FjBcbmwE8yg         03-11-2012 16:34:20,730
QInfo           Worker[8]Eq72thlSdwP5iliFPnavoolWp0FdHYLMqYS8qL         03-11-2012 16:34:20,723
QInfo           Worker[0]A8Hhl7HFQUn3sm3KFP7P0cBg>ZZMSPZVbd5Kig         03-11-2012 16:34:20,667
QTrace          Worker[2]rm<BLV5imARsNTj9GQrP0PyV7iKIJccdrZO3QG         03-11-2012 16:34:20,660
```

Logs loaded: 100. QDebug: 13. QInfo: 29. QCritical: 14. QTrace: 10. QWarn: 14. QLogSampleArea: 11. QError: 9. Total logs in the database: 242.

If you click on the "Help → About" menu button, then you will find out all key shortcuts listed that will allow you to speed up the process of logs browsing.

There are quite a bit of filtering options in QLog Browser, that you can apply in order to find logs that you are interested in. You can specify the custom areas for which you want to read logs or specify additional filters limiting results to the ones with the particular SessionId, ThreadId etc. you can find all filters in the left part of the main window and under the "Options" menu button.

QLog Browser allows you to perform a quick clean up in the database by selecting "File → Clean logs from storage...".

Application doesn't require any installation and it remembers all your connection settings so you don't have to provide it each time when you want to browse logs.

8 Additional utilities

QLog offers an additional utility available with the tool QLog.Utils that you may use in your application. It contains the method for logs cleanup:

```
public void CleanLogsOlderThan(int noDays)
```

It cleans the particular logs if they are older then the provided number of days.

It is recommended that you design a worker project for your application that will perform clean up on all of the logs. All you have to do is to write a simple instruction in worker code, like:

```
QLog.Logger.CleanLogsOlderThan(60);
```

10 Logger methods list

QLog.Logger contains the following public methods:

```
void LogTrace(string msg)
void LogTrace(string msg, params object[] args)
void LogTrace(Exception e)
void LogTrace(string msg, Exception e)
void LogDebug(string msg)
void LogDebug(string msg, params object[] args)
void LogDebug(Exception e)
void LogDebug(string msg, Exception e)
void LogInfo(string msg)
void LogInfo(string msg, params object[] args)
void LogInfo(Exception e)
void LogInfo(string msg, Exception e)
void LogWarn(string msg)
void LogWarn(string msg, params object[] args)
void LogWarn(Exception e)
void LogWarn(string msg, Exception e)
void LogError(string msg)
void LogError(string msg, params object[] args)
void LogError(Exception e)
void LogError(string msg, Exception e)
void LogCritical(string msg)
void LogCritical(string msg, params object[] args)
void LogCritical(Exception e)
void LogCritical(string msg, Exception e)
void Log<Area>(string msg) where Area : QArea
void Log<Area>(string msg, params object[] args) where Area : QArea
void Log<Area>(Exception e) where Area : QArea
void Log<Area>(string msg, Exception e) where Area : QArea
void Flush()
void Flush(bool async)
```

QLog.Utils contains the following public methods:

```
void CleanLogsOlderThan(int noDays)
```