Platform SDK Developer's Guide

Using the Log Library

4/8/2026

Using the Log Library

Java

The purpose of the Platform SDK Log Library is to present an easy-to-use API for logging messages in custom-built applications. Depending on how you configure your logger, you can quickly and easily have log messages of different verbose levels written to any of the following targets:

- Genesys Message Server
- Console
- Specified log files

This document provides some key considerations about how to configure and use this component, as well as code examples that will help you work with the Platform SDK Log Library in your own projects.

Introduction to Loggers

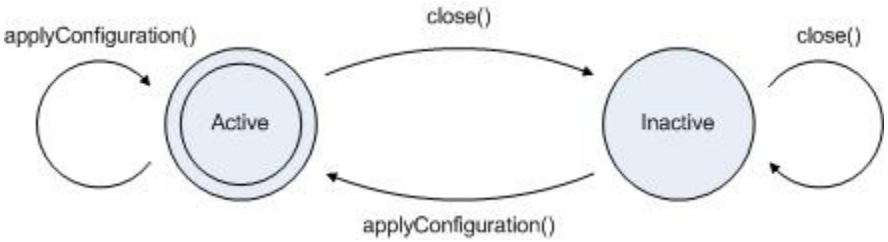
When working with custom Genesys loggers, the first step is to understand the basic process of creating and maintaining your logger. How do you create a logger instance? What configuration options should you use, and how and when can those options be changed? What is required to clean up once the logger is no longer useful?

Luckily, the main functions and lifecycle of a logger are easy to understand. The following list outlines the basic process required to create and maintain your customized logger. For more detailed information, check the *Lifecycle of a Logger section*, or the specific code examples related to *Creating a Logger*, *Customizing your Logger*, *Using your Logger*, or *Cleaning Up Your Code*.

1. Use the `LoggerFactory` to create a `RootLogger` instance.
2. Reconfigure the default `RootLogger` settings, if desired.
    - Create a `LogConfiguration` instance to enable and configure log targets. Depending on the setting you assign, log messages can be sent to the Console, to a Genesys Message Server, or to one or more user-defined log files.
    - `LoggerPolicy` property gives you control over how log messages are created and formatted, or allows you to overwrite `MessageServerProtocol` properties.
3. Use your logger for logging messages.
4. Dispose of the `RootLogger` instance when it is no longer needed. (You can also close the logger if it will be reused in the future.)

Lifecycle of a Logger

There are two possible states for a logger, as shown in the lifecycle diagram below.



Your logger begins in the active state once it is created. If you did not specify any configuration options during creation, then all messages with a verbose level at least equal to `VerboseLevel.Trace` are logged to the Console by default.

You can use the `applyConfiguration` method to change logger configuration settings from either an active or inactive state:

- If the logger is active when you call this method, then all messages being processed will be handled before the logger is stopped and reconfigured. Note that any file targets (from both the old and new configurations) are not closed automatically when the logger is reconfigured, although file targets *can* be closed and a new log file segment started if the new `Segmentation` settings require this.
- If the logger is inactive when you call this method, then it is activated after the new configuration settings are applied.

You can use the `close` method to make the logger inactive without disposing of it. All messages being processed when this method is called are processed before the logger is stopped. Once the logger is inactive, no further messages are processed until after the `applyConfiguration` method is called.

> **Important**
>
> If your logger is connected to Message Server, the logger does not manage the lifecycle of the `MessageServerProtocol` instance. You must manage and close that connection manually.

Creating a Default Logger

This section provides simple code examples that show how to quickly create a logger with the default configuration and use it as part of your application.

Creating a Logger

As always, the starting point is ensuring that you have the necessary Platform SDK libraries referenced and declared in your project. For logging functionality described in this article, that

includes the following packages:

- import com.genesyslab.platform.commons.log.*;
- import com.genesyslab.platform.commons.collections.KeyValueCollection;
- import com.genesyslab.platform.management.protocol.messageserver.LogLevel;
- import com.genesyslab.platform.logging.*;
- import com.genesyslab.platform.logging.configuration.*;
- import com.genesyslab.platform.logging.utilities.*;
- import com.genesyslab.platform.logging.runtime.LoggerException;

Once your project is properly configured and coding about to begin, the first task is to create an instance of the `RootLogger` class. This is easy to accomplish with help from the `LoggerFactory` - the only information you need to provide is a logger name that can be used later to configure targets for filtering logging output.

```
[Java]

try{
        // Create a logger instance:
        RootLogger logger = new LoggerFactory("myLoggerName").getRootLogger();
}
catch(LoggerException e){
        // Handle exceptions...
}
```

The default behavior for this logger is to send all messages of `Trace` verbose and higher to the Console. You can change this behavior by using a `LogConfiguration` instance to change configuration settings and then applying those values to the `RootLogger` instance, as shown below, but for now we will accept the default values.

Using your Logger

With a logger created and ready for use, the next step is to generate some custom log messages and ensure that your logger is working correctly.

One way to generate log messages is with the `write` method. Depending on the parameters used with this method, message formatting can either be provided by templates extracted from LMS files, or through the settings that you configure in a `LogEntry` instance. For the example below, the only formatting come from the `LogEntry` parameter.

```
[Java]

LogEntry logEntry;
logEntry = new LogEntry("Sample Internal message.");
logEntry.setId(CommonMessage.GCTI_INTERNAL.getId());
logger.write(logEntry);

logEntry.setMessage("Sample Debug message.");
logEntry.setId(CommonMessage.GCTI_DEBUG.getId());
logger.write(logEntry);
```

You can also generate log messages by using one of the methods listed in the following table.

Message Level	Available Methods
Debug	• debug(Object arg0) • debug(Object arg0, Throwable arg1) • debugFormat(String arg0, object arg1)
Info	• info(Object arg0) • info(Object arg0, Throwable arg1) • infoFormat(String arg0, object arg1)
Interaction	• warn(Object arg0) • warn(Object arg0, Throwable arg1) • warnFormat(String arg0, object arg1)
Error	• error(Object arg0) • error(Object arg0, Throwable arg1) • errorFormat(String arg0, object arg1)
Alarm	• fatalError(Object arg0) • fatalError(Object arg0, Throwable arg1) • fatalErrorFormat(String arg0, object arg1)

These methods do not use any external templates or formatting, relying entirely on the information passed into them. In the examples below, the messages are logged at `Info` and `Debug` verbose levels, without any changes or formatting.

```
[Java]

logger.info("Sample Info message.");
logger.debug("Sample Debug message.");
```

Cleaning Up Your Code

Once you have finished logging messages with your logger, there are two options available: you can close the logger if you want it to be available for reuse later, or dispose of the logger if your application doesn't need it any longer. (Note that you do not have to close a logger before disposing of it.)

Once closed, a logger remains in an inactive state until either the `ApplyConfiguration` method is called or you dispose of the object, as shown in the *Lifecycle of a Logger* diagram above.

```
[Java]
```

```
// closing the logger
logger.close();
// disposing of the logger
logger = null;
```

Customizing your Logger

Now that you know how to create and use a generic logger, it is time to look at some of the configuration options available to alter the behavior of your logger.

The `LogConfiguration` class allows you change application details, specify targets (including Genesys Message Server) for your log messages, and adjust the verbose level you want to report on. You can apply these changes to either a new logger that is created with the `LogFactory`, or to an existing logger by using the `ApplyConfiguration` method.

> **Tip**
>
> `MessageHeaderFormat` property in the `LogConfiguration` class has no effect on records in the Message Server Database due to message server work specificity.

Creating a LogConfiguration to Specify Targets and Verbose Levels

The first step to configuring the settings for your logger is creating an instance of the `LogConfigurationImpl` class and setting some basic parameters that describe your application.

```
[Java]

LogConfigurationImpl logConfigImpl = new LogConfigurationImpl();
logConfigImpl.setApplicationHost("myHostname");
logConfigImpl.setApplicationName("myApplication");
logConfigImpl.setApplicationId(10);
logConfigImpl.setApplicationType(20);
logConfigImpl.setVerbose(VerboseLevel.ALL);
```

Additional `LogConfiguration` properties that can be configured to specify the name of an application-specific LMS file (`MessageFile`) and whether timestamps should use local or UTC format (`TimeUsage`). These steps aren't shown here for brevity; refer to the API Reference for details.

> **Tip**
>
> If logging to the network, timestamps for log entries always use UTC format to avoid confusion. In this case the `TimeUsage` setting specified by your `LogConfiguration` is ignored.

The next step is to assign this implementation to an actual `LogConfiguration` instance. Once you do that, you can specify the target locations where log messages will be sent and the verbose levels accepted by each individual target. (Only messages with a level greater than or equal to the verbose

setting will be logged.)

```
[Java]

LogConfiguration config = logConfigImpl;

// configure logging to console
config.getTargets().getConsole().setEnabled(true);
config.getTargets().getConsole().setVerbose(VerboseLevel.TRACE);

// configure logging to system events log
config.getTargets().getNetwork().setEnabled(true);
config.getTargets().getNetwork().setVerbose(VerboseLevel.STANDARD);
```

Adding files to your logger requires one extra step: creating and configuring a FileConfiguration instance that provides details about each log file to be used. For example:

```
[Java]

// add logging to Log file "Log\fulllog" - for all messages
FileConfiguration file = new FileConfigurationImpl(true, VerboseLevel.ALL, "Log/fulllogfile");
file.setMessageHeaderFormat(MessageHeaderFormat.FULL);
config.getTargets().getFiles().add(file);

// add logging to Log file "Log\infolog" - for Info (and higher) messages
file = new FileConfigurationImpl(true, VerboseLevel.TRACE, "Log/infologfile");
file.setMessageHeaderFormat(MessageHeaderFormat.SHORT);
config.getTargets().getFiles().add(file);
```

> ### Warning
> Each file added as a target must have a unique name. If two or more items are added to the file collection with the same name, only one file target will be created with the lowest specified verbose level. Other settings will be taken from one of the items using the same filename, but there is no way to predict which item will be used.

In the example above, the first line of code ensures that your logger will process messages for all verbose levels - but each target location has its own setting afterwards that specifies what level of messages can be logged by that target. You also can enable or disable individual logging targets by changing and then reapplying the settings in the `LogConfiguration` instance.

Once you have created and configured the `LogConfiguration` instance, all that remains is to apply those settings to your logger. The following code shows how you can apply these settings to either a new Logger instance, or an already existing logger.

```
[Java]

// applying new configuration to an existing logger
logger.applyConfiguration(config);
```

For more information about using `ApplyConfiguration`, see the *Lifecycle of a Logger* section above and the API Reference entry for that method.

Alternative Ways to Create a LogConfiguration

Another way to create a `LogConfiguration` instance is by parsing a `KeyValueCollection` that contains the appropriate settings. A brief code example of how to accomplish this is provided below.

```
[Java]

KeyValueCollection kvConfig = new KeyValueCollection();
// verbose level of logger will be VerboseLevel.All
kvConfig.addString("verbose","all");

// enable output of info (and higher) messages to console
kvConfig.addString("trace","stdout");

// add file target for debug debug output
kvConfig.addString("debug","Log/dbglogfile");

// Parse the created keyValueCollection. Messages generated during parsing are logged to
Console.
LogConfiguration config = LogConfigurationFactory.parse(kvConfig, (ILogger)new
Log4JLoggerFactoryImpl ());
```

Finally, you can also create a `LogConfiguration` by parsing an `org.w3c.dom.Element` that contains the appropriate settings. This Element can be created manually, or obtained from a `CfgApplication` object.

Dealing with Sensitive Log Data

There are two optional filters included as part of the common Platform SDK functionality that can be used to handle senstive log data. These are not part of the Log Library but are discussed here to help ensure sensitive data is properly considered and handled in any custom applications involving logging.

- Hiding Data in Logs - The first option to protect sensitive data is to prevent it from being printed to log files at all.
- Adding Predefined Prefix/Postfix Strings - The second option does not hide the sensitive information directly, but adds user-defined strings around values for selected key-value pairs. This makes it easy for you to locate and removed sensitive data in case log files need to be shared or distributed for any reason.

For more information about these filters, refer to the `KeyValueOutputFilter` documentation in this API Reference.

Hiding Data in Logs

In the code except below, the `KeyValuePrinter` class is used to hide any value of a key-value pair where the key is "Password":

```
[Java]

KeyValueCollection kvOptions = new KeyValueCollection();
KeyValueCollection kvData = new KeyValueCollection();
kvData.addString("Password", KeyValuePrinter.HIDE_FILTER_NAME);
```

```
KeyValuePrinter hidePrinter = new KeyValuePrinter(kvOptions, kvData);
KeyValuePrinter.setDefaultPrinter(hidePrinter);

KeyValueCollection col = new KeyValueCollection();
col.addString("Password", "secretPassword");
```

As result, the KeyValueCollection log output will have the "secretPassword" value printed as "*****". Values for other keys will display as usual.

Adding Predefined Prefix/Postfix Strings

The `PrefixPostfixFilter` class is designed to give you the ability to wrap parts of the log with predefined prefix/postfix strings. This makes it possible to easily filter out sensitive information from an already-printed log file when such a necessity arises.

In the code except below, the KeyValuePrinter is set to wrap "Password" key-value pairs in the "<###" (prefix), "###>" (postfix) strings:

```
[Java]

KeyValueCollection kvData = new KeyValueCollection();
KeyValueCollection kvPPfilter = new KeyValueCollection();
KeyValueCollection kvPPOptions = new KeyValueCollection();
kvPPfilter.addString(KeyValuePrinter.CUSTOM_FILTER_TYPE, "PrefixPostfixFilter");
kvPPOptions.addString(PrefixPostfixFilter.KEY_PREFIX_STRING, "<###");
kvPPOptions.addString(PrefixPostfixFilter.VALUE_POSTFIX_STRING, "###>");
kvPPOptions.addString(PrefixPostfixFilter.KEY_POSTFIX_STRING, ">");
kvPPOptions.addString(PrefixPostfixFilter.VALUE_PREFIX_STRING, "<");
kvPPfilter.addList(KeyValuePrinter.CUSTOM_FILTER_OPTIONS, kvPPOptions);
kvData.addList("Password", kvPPfilter);
KeyValuePrinter.setDefaultPrinter(
        new KeyValuePrinter(new KeyValueCollection(), kvData));

KeyValueCollection col = new KeyValueCollection();
col.addString("test", "secretPassword");
```

As result, the `KeyValueCollection` log output will have the "Password-secretPassword" key-value printed as "<###Password-secretPassword###>", leaving all other key-values printed as normal.

.NET

The purpose of the Platform SDK Log Library is to present an easy-to-use API for logging messages in custom-built applications. Depending on how you configure your logger, you can quickly and easily have log messages of different verbose levels written to any of the following targets:

- Genesys Message Server
- Console
- .NET Trace
- Windows System Log (Application Log only)
- Specified log files

This document provides some key considerations about how to configure and use this component, as well as code examples that will help you work with the Platform SDK Log Library for .NET in your own projects.

Introduction to Loggers

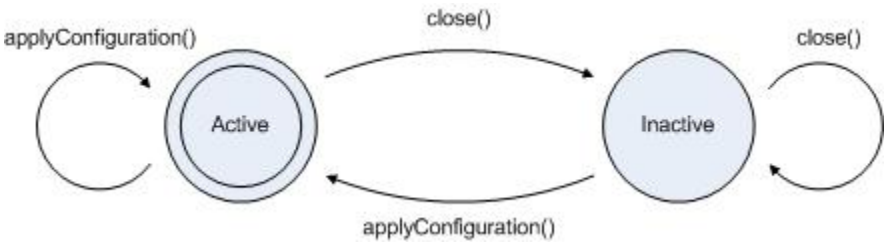
When working with custom Genesys loggers, the first step is to understand the basic process of creating and maintaining your logger. How do you create a logger instance? What configuration options should you use, and how and when can those options be changed? What is required to clean up once the logger is no longer useful?

Luckily, the main functions and lifecycle of a logger are easy to understand. The following list outlines the basic process required to create and maintain your customized logger. For more detailed information, check the *Lifecycle of a Logger section*, or the specific code examples related to *Creating a Logger*, *Customizing your Logger*, *Using your Logger*, or *Cleaning Up Your Code*.

1. Use the LoggerFactory to create an ILogger instance.
2. Reconfigure the default ILogger settings, if desired.
   - NetworkProtocol property allows you to specify a MessageSeverProtocol instance. This will let your ILogger instance send messages to Genesys Message Server.
   - LoggerPolicy property gives you control over how log messages are created and formatted, or allows you to overwrite MessageServerProtocol properties.
   - LogConfiguration class allows you to configure other aspects of your ILogger instance, which are applied with the ApplyConfiguration method.
3. Use the logger for logging messages.
4. Dispose of the ILogger instance when it is no longer needed. (You can also close the logger if it will be reused in the future.)

Lifecycle of a Logger

There are two possible states for a logger, as shown in the lifecycle diagram below.



Your logger begins in the active state once it is created. If you did not specify any configuration options during creation, then all messages with a verbose level at least equal to `VerboseLevel.Trace` are logged to the Console by default.

You can use the `ApplyConfiguration` method to change logger configuration settings from either an active or inactive state:

- If the logger is active when you call this method, then all messages being processed will be handled before the logger is stopped and reconfigured. Note that any file targets (from both the old and new configurations) are not closed automatically when the logger is reconfigured, although file targets can be closed and a new log file segment started if the new `Segmentation` settings require this.
- If the logger is inactive when you call this method, then it is activated after the new configuration settings are applied.

You can use the `Close` method to make the logger inactive without disposing of it. All messages being processed when this method is called are processed before the logger is stopped. Once the logger is inactive, no further messages are processed until after the `ApplyConfiguration` method is called.

> **Important**
>
> If your logger is connected to Message Server, the logger does not manage the lifecycle of the `MessageServerProtocol` instance. You must manage and close that connection manually.

Creating a Default Logger

This section provides simple code examples that show how to quickly create a logger with the default configuration and use it as part of your application.

Creating a Logger

As always, the starting point is ensuring that you have the necessary Platform SDK libraries referenced and declared in your project. For logging functionality, that includes the following namespaces:

- `Genesyslab.Platform.Commons.Logging`
- `Genesyslab.Platform.Logging`
- `Genesyslab.Platform.Logging.Configuration`
- `Genesyslab.Platform.Logging.Utilities`

Once your project is properly configured and coding about to begin, the first task is to create an instance of the `ILogger` class. This is easy to accomplish with help from the `LoggerFactory` - the only information you need to provide is a logger name that can be used later to configure targets for filtering logging output.

```
[C#]

IRootLogger logger = LoggerFactory.CreateRootLogger("myLoggerName");
```

The default behavior for this logger is to send all messages of `Trace` verbose and higher to the

Console. You can change this behavior by using the `ILogConfiguration` interface to pass configuration settings into the `LoggerFactory`, as shown below, but for this example we will accept the default values.

Using your Logger

Now that your logger is created and ready for use, the next step is to generate some custom log messages and ensure that the logger is working correctly.

One way to generate log messages is with the `Write` method. Message formatting is provided either by templates extracted from LMS files or directly from a `LogEntry` parameter, depending on what information you pass into the method. For the example below, LMS file templates provide formatting.

```
[C#]

//log the message with standard id: "9999|STANDARD|GCTI_INTERNAL|Internal error '%s' occurred"
//formatting template is extracted from LMS file
logger.Write((int)CommonMessage.GCTI_INTERNAL, "Sample Internal message.");

//log the message with standard id: "9900|DEBUG|GCTI_DEBUG|%s"
//formatting template is extracted from LMS file
logger.Write((int)CommonMessage.GCTI_DEBUG, "Sample Debug message.");
```

You can also generate log messages by using one of the methods listed in the following table.

Message Level	Available Methods
Debug	• Debug(object message) • Debug(object message, Exception exception) • DebugFormat(string format, params object [] args)
Info	• Info(object message) • Info(object message, Exception exception) • InfoFormat(string format, params object [] args)
Interaction	• Warn(object message) • Warn(object message, Exception exception) • WarnFormat(string format, params object [] args)
Error	• Error(object message) • Error(object message, Exception exception) • ErrorFormat(string format, params object [] args)

Message Level	Available Methods
Alarm	• FatalError(object message) • FatalError(object message, Exception exception) • FatalErrorFormat(string format, params object [] args)

These methods do not use any external templates or formatting, relying entirely on the information passed into them. In the examples below, the messages are logged at `Info` and `Debug` verbose levels, without any changes or formatting.

```
[C#]

logger.Info("Sample Info message.");
logger.Debug("Sample Debug message.");
```

Cleaning Up Your Code

Once you have finished logging messages with your logger, there are two options available: you can close the logger if you want it to be available for reuse later, or dispose of the logger if your application doesn't need it any longer. (Note that you do not have to close a logger before disposing of it.)

Once closed, a logger remains in an inactive state until either the `ApplyConfiguration` or `Dispose` method is called, as shown in the lifecycle diagram above.

```
[C#]

//closing the logger
logger.Close();
...
//disposing of the logger
logger.Dispose();
```

Customizing your Logger

Now that you know how to create and use a generic logger, it is time to look at some of the configuration options available to alter the behavior of your logger.

The `LogConfiguration` class allows you change application details, specify targets (including Genesys Message Server) for your log messages, and adjust the verbose level you want to report on. You can apply these changes to either a new logger that is created with the `LogFactory`, or to an existing logger by using the `ApplyConfiguration` method.

> **Tip**
>
> The `setMessageHeaderFormat` method has no effect on records in the Message Server

> Database due to message server work specificity.

Creating a LogConfiguration to Specify Targets and Verbose Levels

The first step to configuring the settings for your logger is creating an instance of the `LogConfiguration` class and setting some basic parameters that describe your application.

```
[C#]

LogConfiguration config = new LogConfiguration
{
    ApplicationHost = "myHostname",
    ApplicationName = "myApplication",
    ApplicationId = 10,
    ApplicationType = 20
};
```

Additional `LogConfiguration` properties that can be configured to specify the name of an application-specific LMS file (`MessageFile`) and whether timestamps should use local or UTC format (`TimeUsage`). These steps aren't shown here for brevity; refer to the API Reference for details.

> ### Tip
> If logging to the network, as described in *Logging Messages to Genesys Message Server*, timestamps for log entries always use UTC format to avoid confusion. In this case the `TimeUsage` setting specified by your `LogConfiguration` is ignored.

The next step is to specify the target locations where log messages are recorded, and to configure the verbose levels for the logger and for individual targets. (Only messages with a level greater than or equal to the verbose setting will be logged.)

```
[C#]

config.Verbose = VerboseLevel.All;

//configure logging to console
config.Targets.Console.IsEnabled = true;
config.Targets.Console.Verbose = VerboseLevel.Trace;

//configure logging to system events log
config.Targets.System.IsEnabled = true;
config.Targets.System.Verbose = VerboseLevel.Standard;

//add logging to Log file "Log\fulllog" - for all messages
config.Targets.Files.Add(new FileConfiguration(true, VerboseLevel.All, "Log/fulllogfile"));
//add logging to Log file "Log\infolog" - for Info (and higher) messages
config.Targets.Files.Add(new FileConfiguration(true, VerboseLevel.Trace, "Log/infologfile"));
```

In the example above, the first line of code ensures that your logger will process messages for all verbose levels - but each target location has its own setting afterwards that specifies what level of messages can be logged by that target. You also can enable or disable individual logging targets by

changing and then reapplying the settings in the LogConfiguration.

> **Warning**
> Each file added as a target must have a unique name. If two or more items are added to the file collection with the same name, only one file target will be created with the lowest specified verbose level. Other settings will be taken from one of the items using the same filename, but there is no way to predict which item will be used.

Once you have created and configured the LogConfiguration instance, all that remains is to apply those settings to your logger. The following code shows how you can apply these settings to either a new Logger instance, or an already existing logger.

```
[C#]

//applying new configuration to an existing logger
logger.ApplyConfiguration(config);
...
//apply new configuration to a new logger when it is created
IRootLogger newlogger = LoggerFactory.CreateRootLogger("NewLoggerName", config);
```

For more information about using ApplyConfiguration, see the logger lifecycle section above and the API Reference entry for that method.

Alternative Ways to Create a LogConfiguration

Another way to create a LogConfiguration is by parsing a KeyValueCollection that contains the appropriate settings. A brief code example of how to accomplish this is provided below.

```
[C#]

KeyValueCollection kvConfig = new KeyValueCollection();
//verbose level of logger will be VerboseLevel.All
kvConfig.Add("verbose","all");
//enable output of info (and higher) messages to console
kvConfig.Add("trace","stdout");
//add file target for debug debug output
kvConfig.Add("debug","Log/dbglogfile");
//Parse the created keyValueCollection. Messages generated during parsing are logged to
Console.
LogConfiguration config = LogConfigurationFactory.Parse(kvConfig, new ConsoleLogger());
```

Finally, you can also create a LogConfiguration by parsing an XElement that contains the appropriate settings, as shown below.

```
[C#]

XElement xElementConfig =
        new XElement("CfgApplication",
                new XElement("options",
                        new XElement("list_pair",
                                new XAttribute("key","log"),
                                        XElement.Parse("<str_pair key=\"verbose\" value =
\"all\"/>"),
                                        XElement.Parse("<str_pair key=\"trace\" value =
```

```
\"stdout\"/>"),
                                        XElement.Parse("<str_pair key=\"debug\" value = \"Log/
dbglogfile\"/>"),
                        )
                )
        );
LogConfiguration config = LogConfigurationFactory.Parse(xElementConfig, new ConsoleLogger());
```

Although the `XElement` can be created manually (as shown above), it is much more likely that it will be obtained from a `CfgApplication` object. The following code example illustrates how this can be done.

```
[C#]

ConfService confservice=null;
//...
//initializing the ConfService
//...
CfgApplication cfgApp = confservice.RetrieveObject<CfgApplication>(
        new CfgApplicationQuery{Name = "Sample Application"});
XElement xElementConfig = cfgApp.ToXml();
LogConfiguration config = LogConfigurationFactory.Parse(xElementConfig, new ConsoleLogger());
```

Logging Messages to Genesys Message Server

Creating a connection with Genesys Message Server is similar to setting other targets for your logger, but contains a couple of additional steps. Several new settings are required to determine how your logger handles buffering and spooling when sending log messages over the network. Once that is complete, you also have to create (and manage) a protocol object that connects to Message Server.

The following example shows how this can be accomplished. For details and additional information about the properties being configured, refer to the appropriate API Reference entries.

```
[C#]

LogConfiguration config = new LogConfiguration {Verbose = VerboseLevel.All};
config.Targets.Network.IsEnabled = true;
config.Targets.Network.Verbose = VerboseLevel.All;
config.Targets.Network.Buffering = Buffering.On|Buffering.KeepOnProtocolChange;
config.Targets.Network.SpoolFile = "temp/spool";

//create and open connection to message server
MessageServerProtocol protocol = new MessageServerProtocol(
        new Endpoint(myApplication, myHostname, myPort));
protocol.Open();

IRootLogger logger = LoggerFactory.CreateRootLogger("mySample");
logger.NetworkProtocol = protocol;
logger.ApplyConfiguration(config);
```

It is important to remember that any connection created to Message Server is not managed automatically by the Logger lifecycle. You are responsible to manage and dispose of the connection manually.

Dealing with Sensitive Log Data

There are two optional filters included as part of the common Platform SDK functionality that can be used to handle sensitive log data. These are not part of the Log Library for .NET, but are discussed here to help ensure sensitive data is properly considered and handled in any custom applications involving logging.

- Hiding Data in Logs - The first option to protect sensitive data is to prevent it from being printed to log files at all.
- Adding Predefined Prefix/Postfix Strings - The second option does not hide the sensitive information directly, but adds user-defined strings around values for selected key-value pairs. This makes it easy for you to locate and removed sensitive data in case log files need to be shared or distributed for any reason.

Hiding Data in Logs

In the code except below, the `KeyValuePrinter` class is used to hide any value of a key-value pair where the key is "Password":

```
[C#]

KeyValueCollection kvOptions = new KeyValueCollection();
KeyValueCollection kvData = new KeyValueCollection();
kvData["Password"] = KeyValuePrinter.HideFilterName;
KeyValuePrinter hidePrinter = new KeyValuePrinter(kvOptions, kvData);
KeyValuePrinter.DefaultPrinter = hidePrinter;

KeyValueCollection col = new KeyValueCollection();
col["Password"] = "secretPassword";
```

As result, the `KeyValueCollection` log output will have the "secretPassword" value printed as "*****". Values for other keys will display as usual.

Adding Predefined Prefix/Postfix Strings

The `PrefixPostfixFilter` class is designed to give you the ability to wrap parts of the log with predefined prefix/postfix strings. This makes it possible to easily filter out sensitive information from an already-printed log file when such a necessity arises.

In the code except below, the `KeyValuePrinter` is set to wrap "Password" key-value pairs in the "<###" (prefix), "###>" (postfix) strings:

```
[C#]

KeyValueCollection kvData = new KeyValueCollection();
KeyValueCollection kvPPfilter = new KeyValueCollection();
KeyValueCollection kvPPOptions = new KeyValueCollection();
kvPPfilter[KeyValuePrinter.CustomFilterType] = typeof(PrefixPostfixFilter).FullName;
kvPPOptions[PrefixPostfixFilter.KeyPrefixString] = "<###";
kvPPOptions[PrefixPostfixFilter.ValuePrefixString] = "<";
kvPPOptions[PrefixPostfixFilter.ValuePostfixString] = "###>";
kvPPOptions[PrefixPostfixFilter.KeyPostfixString] = ">";
kvPPfilter[KeyValuePrinter.CustomFilterOptions] = kvPPOptions;
kvData["Password "] = kvPPfilter;
```

```
KeyValuePrinter.DefaultPrinter = new KeyValuePrinter(new KeyValueCollection(), kvData);
KeyValueCollection col = new KeyValueCollection();
col["Password"] = "myPassword";
```

As result, the `KeyValueCollection` log output will have the "Password-secretPassword" key-value printed as "<###Password-secretPassword###>", leaving all other key-values printed as normal.