Platform SDK Developer's Guide

Setting up Logging in Platform SDK

4/20/2025

Setting up Logging in Platform SDK

Logging for Java

Setting up log4j Logging

The easiest way to set up Platform SDK logging in Java is to use the built-in integration with log4j. There are two possible ways to do this:

- Using code, by creating a `Log4JLoggerFactoryImpl` instance and setting it as the global logger factory for Platform SDK at the beginning of your program, like this:

```
com.genesyslab.platform.commons.log.Log.setLoggerFactory(new Log4JLoggerFactoryImpl());
```

Or:

- Using a Java system variable, by setting `com.genesyslab.platform.commons.log.loggerFactory` to the fully qualified name of the `ILoggerFactory` implementation class. For example, to set up log4j as the logging implementation you can start your application using the following command:

```
java
-Dcom.genesyslab.platform.commons.log.loggerFactory=com.genesyslab.platform.commons.log.Log4JLoggerFactoryImpl
<MyMainClass>
```

Providing a Custom Logging Implementation

If log4j does not fit your needs, it is also possible to provide your own implementation of logging.

In order to do that, you will need to complete the following steps:

1. Implement the `ILogger` interface, which contains the methods that the Platform SDK uses for logging messages, by extending the `AbstractLogger` class.
2. Implement the `ILoggerFactory` interface, which should create instances of your `ILogger` implementation.
3. Finally, set up your `ILoggerFactory` implementation as the global Platform SDK `LoggerFactory`, as described above.

Setting Up Internal Logging for Platform SDK

To use internal logging in Platform SDK, you have to set a logger implementation in Log class *before* making any other call to Platform SDK. There are two ways to accomplish this:

1. Set the `com.genesyslab.platform.commons.log.loggerFactory` system property to the fully qualified

name of the factory class

2. Use the `Log.setLoggerFactory(...)` method

The only log factory available in Platform SDK itself is `com.genesyslab.platform.commons.log.Log4JLoggerFactoryImpl` which uses log4j. You will have to setup log4j according to your needs, but a simple log4j configuration file is shown below as an example.

```
log4j.logger.com.genesyslab.platform=DEBUG, A1
log4j.appender.A1=org.apache.log4j.FileAppender
log4j.appender.A1.file=psdk.log
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-4r [%t] %-5p %-25.25c %x - %m%n
```

The easiest way to set system property is to use `-D` switch when starting your application:

```
-Dcom.genesyslab.platform.commons.log.loggerFactory=com.genesyslab.platform.commons.log.Log4JLoggerFactoryImpl
```

Logging with AIL

In Interaction SDK (AIL) and Genesys Desktop applications, you can enable the Platform SDK logs by setting the option `log/psdk-debug = true`.

At startup, AIL calls: `Log.setLoggerFactory(new Log4JLoggerFactoryImpl());`

The default level of the logger `com.genesyslab.platform` is `WARN` (otherwise, applications would be literally overloaded with logs). The option is dynamically taken into account; it turns the logger level to `DEBUG` when set to true, and back to `WARN` when set to false.

Truncating Large Logs Using PSDK.DATA

Starting from PSDK 8.1.1, a special logger was added (in terms of log4j configuration) with the name: `PSDK.DATA`. It was initially designed for the configuration server protocol to:

1. truncate main Platform SDK logs, and
2. allow creation of logs with full protocol data dumps.

A sample of log4j configuration follows:

```
log4j.logger.com.genesyslab.platform=TRACE, A1
log4j.logger.PSDK.DATA=TRACE, A2

log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-4r [%t] %-5p %-25.25c %x - %m%n

log4j.appender.A2=org.apache.log4j.FileAppender
log4j.appender.A2.File=d:\\psdkdata.log
log4j.appender.A2.Append=true
log4j.appender.A2.Threshold=TRACE
log4j.appender.A2.layout=org.apache.log4j.PatternLayout
log4j.appender.A2.layout.ConversionPattern=%-4r [%t] %-5p %-25.25c %x - %m%n
```

> **Tip**
>
> This feature has side effect: when log4j is configured to use "rootLogger" for all logs (including Platform SDK) then it may record protocol messages twice - once for the main logger and again for the data logger.

The goal of the extension is to resolve issues where large log files affect application performance. For example, an application may read a lot of configuration objects and require Platform SDK logging to be enabled. In this case, a configuration protocol message that arrives containing 1 MB of packed data could lead to roughly 6 MB of log data which (in most cases) is not required. These large log records can be truncated, recording enough data to ensure that configuration information is available and that data flow is ok.

In some case, a full data dump may be required in logs. In this case, the truncation enabling parameter is passed to the static context of `ToStringHelper`, which generates a string representation of abstract protocol messages with attributes. Creating an additional logger that manages separated protocol messages with this context may be useful at times, while initializing the `com.genesyslab.platform` logger for general Platform SDK logging without enabling full dumps by default is better in most cases.

Using JVM system properties, which are checked before log record generation and can enable/disable full data dumps, is an alternative way to handle this scenario. It may be a preferred solution, although usage of system properties may not work depending on how application containers are used.

Logging for .NET

Setting up Logging

For .NET development, the `EnableLogging` method allows logging to be easily set up for any classes that implement the `ILogEnabled` interface. This includes:

- All protocol classes: `TServerProtocol, StatServerProtocol,` etc.
- The `WarmStandbyService` class of the Warm Standby Application Block.

For example:

```
tserverProtocol.EnableLogging(new MyLoggerImpl());
```

Providing a Custom Logging Implementation

You can provide your custom logging functionality by implementing the `ILogger` interface. Samples of how to do this are provided in the following section.

Samples

You can download some samples of classes that implement the `ILogger` interface:

- AbstractLogger: This class can make it easier to implement a custom logger, by providing a default implementation of `ILogger` methods.
- TraceSourceLogger: A logger that uses the .NET TraceSource framework. It adapts the Platform SDK logger hierarchy to the non-hierarchical TraceSource configuration.
- Log4netLogger: A logger that uses the log4net libraries.