Platform SDK Developer's Guide

Setting up Logging in Platform SDK

12/13/2025

Setting up Logging in Platform SDK

Java

Using the Built-In Logging Implementation

The Platform SDK Commons library provides adapters for the following implementations:

- com.genesyslab.platform.commons.log.SimpleLoggerFactoryImpl - redirect Platform SDK logs to System.out;
- com.genesyslab.platform.commons.log.JavaUtilLoggerFactoryImpl - redirect Platform SDK logs to Java common java.util.logging logging system;
- com.genesyslab.platform.commons.log.Log4JLoggerFactoryImpl - redirect Platform SDK logs to underlying Log4j 1.x;
- com.genesyslab.platform.commons.log.Log4J2LoggerFactoryImpl - redirect Platform SDK logs to underlying Log4J 2;
- com.genesyslab.platform.commons.log.Slf4JLoggerFactoryImpl - redirect Platform SDK logs to underlying Slf4j.

Note: Prior to release 8.5.102.02, the only log adapter available was for log4j v1.x and short names were not available.

By default, these log implementations are switched off but you can enable logging by using one of the methods described below.

1. In Your Application Code

The easiest way to set up Platform SDK logging in Java is in your code, by creating a factory instance for the log adapter of your choice and set it as the global logger factory for Platform SDK at the beginning of your program. An example using the log4j 1.x adapter is show here:

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

2. Using a Java System Variable

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=<log_type> <MyMainClass>
```

Where <log_type> is either a full-defined class names with packages, or one of the following short names:

- console - for SimpleLoggerFactoryImpl (to System.out);

- jul - for JavaUtilLoggerFactoryImpl;
- log4j - for the Log4J 1.x adaptor;
- log4j2 - for the Log4J 2 adaptor;
- slf4j - for the Slf4j adaptor;
- auto - with this value, Platform SDK Commons logging tries to detect available the logging system from the list of ['Log4j2', 'Slf4j', 'Log4j']; if no log system from the list is detected then the `JavaUtilLoggerFactoryImpl` adapter will be used.

3. Configuration in the Class Path

You can also configure logging using a `PlatformSDK.xml` Java properties file that is specified in your class path:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
  <entry
key="com.genesyslab.platform.commons.log.loggerFactory">com.genesyslab.platform.commons.log.Log4JLoggerFactoryI
</properties>
```

For more information, refer to details about the `PsdkCustomization` class in the API Reference Guide.

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

One of the log factories 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 literally be 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.

Dedicated loggers

Platform SDK has several specialized loggers:

1. com.genesyslab.platform.ADDP
2. com.genesyslab.platformmessage.request
3. com.genesyslab.platformmessage.receive

Dedicated ADDP Logger

ADDP logs can be enabled using common Platform SDK log configuration.

```
log4j.logger.com.genesyslab.platform=INFO, 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
```

In addition, the `com.genesyslab.platform.ADDP` logger is controlled by the `addp-trace` option. If ADDP log is not required on INFO level, it can be disabled using the following option:

```
PropertyConfiguration config = new PropertyConfiguration();
config.setAddpTraceMode(AddpTraceMode.None);
```

or

```
config.setAddpTraceMode(AddpTraceMode.Remote);
```

The `addp-trace` option has no effect when DEBUG level is set. ADDP logs will be printed regardless of the option value.

> **Important**
>
> In Platform SDK 8.5.0, the second ADDP logger (`AddpIntreceptor`) was removed to avoid ADDP log duplication when `RootLogger` of the logging system is set to DEBUG level.

Instead of using second ADDP logger to print logs to another file, it is possible to specify additional appender.

A sample configuration is provided below:

```
log4j.logger.com.genesyslab.platform=WARN, A1
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-d [%t] %-5p %-25.25c %x - %m%n
log4j.appender.A1.Threshold=WARN

//additional log file with addp traces.
log4j.logger.com.genesyslab.platform.ADDP=INFO, A2
log4j.appender.A2=org.apache.log4j.FileAppender
log4j.appender.A2.file=addp.log
log4j.appender.A2.append=false
log4j.appender.A2.layout=org.apache.log4j.PatternLayout
log4j.appender.A2.layout.ConversionPattern=%-d [%t] %-5p %-25.25c %x - %m%n
```

Dedicated Request and Receive Loggers

A sample Log4j configuration is shown here:

```
log4j.logger.com.genesyslab.platformmessage.request=DEBUG, A1
log4j.logger.com.genesyslab.platformmessage.receive=DEBUG, A1
```

> **Important**
>
> In PSDK 8.5.0 version the `PSDK.DATA` logger was replaced with `com.genesyslab.platformmessage.request` and `com.genesyslab.platformmessage.receive` loggers.

These loggers allow printing complete message attribute values. By default, large attribute logs are truncated to avoid application performance impact:

```
'EventInfo' (2) attributes:
        VOID_DELTA_VALUE [bstr] =
           0x00 0x01 0xFF 0xFF 0x00 0x05 0x00 0x00 0x00 0x00
           0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
           0x09 0x00 0x00 0x00 0x05 0x00 0x00 0x00 0x00 0x00
           ... [output truncated, 362 bytes left out of 512]
```

However, in some cases a full data dump may be required in logs. There are three possible ways to do this, as shown below:

> **Important**
>
> To avoid log duplication when the logging system `RootLogger` is configured to DEBUG level, these loggers are disabled by default and can be activated with a system property. This system property affects both loggers.

1. Activate using system properties:

```
-Dcom.genesyslab.platform.trace-messages=true //for all protocols
-Dcom.genesyslab.platform.Reporting.StatServer.trace-messages=true //only for stat protocol
```

2. Activate from code:

```
//for all protocols
PsdkCustomization.setOption(PsdkOption.PsdkLoggerTraceMessages, "false");

//only for stat protocol
String protocolName = StatServerProtocolFactory.PROTOCOL_DESCRIPTION.toString();
PsdkCustomization.setOption(PsdkOption.PsdkLoggerTraceMessages, protocolName, "true");
```

These static options should be set once at the beginning of the program, before opening Platform SDK protocols.

3. Activate from `PlatformSDK.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
  <entry key="com.genesyslab.platform.trace-messages">true</entry>
</properties>
```

For details about the `PsdkCustomization` class, refer to the API Reference Guide.

.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.