Platform SDK Developer's Guide

Additional Logging Features

12/17/2025

Additional Logging Features

Java

Application Configuration Manager Component

The Application Configuration Manager component is a new addition to the Application Template Application Block.

This component monitors the application configuration from Configuration Server and provides notification of any updates to options for your custom application, options of connected servers, or options of their host objects. It also checks the availability of Log4j2 logging framework and automatically enables Log4j2 configuration based on the application logging options in Configuration Manager.

The quickest way to get an application configured for logging in accordance to the application "log" section might look like the following example:

```
public class MyApplication {

  protected static final LmsEventLogger LOG =
LmsLoggerFactory.getLogger(MyApplication.class);
.....
    GFApplicationConfigurationManager appManager =
            GFApplicationConfigurationManager.newBuilder()
            .withCSEndpoint(new Endpoint("CS-primary", csHost1, csPort1))
            .withCSEndpoint(new Endpoint("CS-backup", csHost2, csPort2))
            .withClientId(clientType, clientName)
            .withUserId(csUsername, csPassword)
            .build();
    appManager.register(new GFAppCfgOptionsEventListener() {
        public void handle(final GFAppCfgEvent event) {
            Log.getLogger(getClass()).info(
                    "The application configuration options received: " + event);
            // Init or update own application options from 'event.getAppConfig()'
        }});
    appManager.init();

    // LmsEventLogger method usage:
    LOG.log(CommonLmsEnum.GCTI_APP_INIT_COMPLETED);
    // Common ILogger method usage:
    LOG.info("Some Log4j2 info message");
.....
    // Shutdown the configuration manager:
    appManager.done();
```

In this example, the builder for the manager creates and initializes an internal instance of `ConfService` and encapsulates the `WarmStandby` service to handle failures of the Configuration Server connection.

Created with these parameters, ConfService has:

- the default ConfService cache enabled;
- a WarmStandby service with default configuration for the two given Configuration Server endpoints;
- automatic Configuration Server subscription for notifications on the application and host object types;

If your application needs to have a custom ConfService instance with a specific configuration (for example, a customized cache) then it is possible to create an Application Configuration Manager on your pre-configured ConfService instance. In this case the manager does not take care of the service connection state or caching - it is up to your application to create and manage the WarmStandby service, Configuration Server subscriptions, and the ConfService cache.

An example of working with a custom ConfService instance is provided below:

```
GFApplicationConfigurationManager appManager =
        GFApplicationConfigurationManager.newBuilder()
        .withConfService(confService)
        .build();
appManager.register(new GFAppCfgOptionsEventListener() {
    public void handle(final GFAppCfgEvent event) {
        Log.getLogger(getClass()).info(
                "The application configuration options received: " + event.getAppConfig());
    }});
appManager.init();
```

Common Logging Interfaces Usage

Platform SDK for Java has its own interface for logging (using com.genesyslab.platform.commons.log) that includes the following classes/interfaces:

- Log,
- ILoggerFactory,
- ILogger

Platform SDK uses the ILogger interface to generate Platform SDK internal log messages, and custom applications are also able to use this logging interface as shown in below:

```
public class SomeUserClass {
    protected static final ILogger log = Log.getLogger(SomeUserClass.class);

    public void doSomething() {
        try {
            log.debug("doing something");
            // ...
        } catch (final Exception ex) {
            log.debug("exception while doing something", ex);
        }
    }
}
```

In this sample, the commons logging messages will go to the particular ILogger interface implementation.

Up to release 8.5.0 of Platform SDK, there were two implementations of the interface available: a silent "NullLogger" (default) and Log4j adapter.

Starting with release 8.5.1, the following additional implementations have been added:

- "simple" console printing implementation;
- java.util.logging adapter;
- Slf4j interface adapter;
- Log4j 2.x adapter.

Also it is possible to create a custom implementation of `ILogger` and `ILoggerFactory`, enable their usage, and get into some other logging system.

LMS Event Loggers and LMS files support

`LmsEventLogger` is an extension of the common Platform SDK `ILogger` interface that is used for logging Genesys LMS events to Message Server or for writing log files in the Genesys-specific format.

An example of simple `LmsEventLogger` usage is provided below:

```
class SampleClass {
    protected final static LmsEventLogger LOG = LmsLoggerFactory.getLogger(SampleClass.class);
    public void method() {
        // Use logger to generate event:
        LOG.log(LogCategory.Application, CommonLmsEnum.GCTI_LOAD_RESOURCE, "users.db", "no
such file");
            // => "Unable to load resource 'users.db', error code 'no such file'"

        // or, for event GCTI_CFG_APP[6053, STANDARD, "Configuration for application
obtained"]:
        LOG.log(CommonLmsEnum.GCTI_CFG_APP);
          // or
        LOG.log(6053); // => "Configuration for application obtained"

        // or "plain" logging methods:
        try {
            LOG.debug("Starting cache load...");
            // ... do something ...
        } catch (final Exception exception) {
            LOG.error("Failed to load cache", exception);
        }
.....
```

The Application Configuration Manager component included with Application Template Application Block is able to automatically configure and keep an updated `LmsMessageConveyor` with LMS files configuration.

However, if your custom application does not use the Application Configuration Manager component then it can configure LMS file usage in the following way:

```
public class SomeUserClass {
    protected static final LmsEventLogger LOG =
LmsLoggerFactory.getLogger(SomeUserClass.class);
```

```
    public void configureLogging() {
        // Create new instance of LMS conveyor:
        LmsMessageConveyor lmsConveyor = new LmsMessageConveyor();

        // Configure it:
        lmsConveyor.loadConfiguration(appConfig);
            // or:
            lmsConveyor.loadConfiguration("MyApp.lms");

        // Reinitialize the LmsLoggerFactory singleton with the new conveyor:
        LmsLoggerFactory.createInstance(lmsConveyor);
            // or
            LmsLoggerFactory.setLoggerFactoryImpl(Log.LOG_FACTORY_LOG4J2, lmsConveyor);
    }
.....
```

The LMS loggers also have several implementations in order to support different target logging frameworks including java.util.logging, log4j v1, slf4j, and log4j v2. You can enable specific target framework usage synchronously with the Platform SDK common logging.

The AppTemplate application block contains the LMS event loggers, a "common.lms" file, its correspondent `CommonLmsEnum` class, and an `LmsEnum` generator tool for generation of specific enumerations from the LMS files for your custom applications.

Using Custom LMS Files and Correspondent LmsEnums

1. Create a custom LMS file with the default localization context, for example, "MyApp.lms":

   ```
   xxxxxxxx|LMS|1.0|MyApp.lms|8.5.100|*

   21001|STANDARD|MY_APP_START_EVENT|Application '%s' started the work
   21002|ALARM|MY_APP_DATABASE_LOST|App '%s' failed to connect to database '%s'
   .....
   ```

2. Generate the corresponding enumeration class using the Platform SDK generator:

   ```
   %> java -cp apptemplate.jar
   com.genesyslab.platform.apptemplate.lmslogger.LmsEnumGenerator MyApp.lms MyAppLmsEnum
   custom.package.name
   ```

   As a result there will be a `custom.package.name.MyAppLmsEnum` enumeration containing declarations from `MyApp.lms`.

3. (Optional) Create a customized version of the `MyApp.lms` file with localized messages.

4. (Optional) Load the enumeration(s) with message conveyor and initialize `LmsLoggerFactory` with it. The Application Template applcation block exposes annotations processor, which collects information about available `LmsEnums` in the application build time, so the default `LmsMessageConveyor()` constructor is able to register generated `LmsEnums` automatically. If it does not do this for some reason, and there is no acceptable way to let the annotation processor work for your use case, then it is possible to initialize `LmsMessageConveyor` explicitly:

   ```
   LmsMessageConveyor lmsConveyor = new LmsMessageConveyor(CommonLmsEnum.class,
   MyAppLmsEnum.class);
   lmsConveyor.loadConfiguration("MyApp.lms"); // optional - if there is a localized LMS
   file

   // Reinitialize the LmsLoggerFactory singleton with the new conveyor:
   ```

```
LmsLoggerFactory.createInstance(lmsConveyor);
    // or
    LmsLoggerFactory.setLoggerFactoryImpl(Log.LOG_FACTORY_LOG4J2, lmsConveyor);
```

5. Call the loggers

```
lmsLog.log(MyAppLmsEnum.MY_APP_START_EVENT, "my application");
lmsLog.log(LogCategory.Alarm, MyAppLmsEnum.MY_APP_DATABASE_LOST, "my application",
"dbname");
```

Enabling Logging Framework Usage

The Platform SDK common loggers and LMS events loggers may be configured to direct logs to particular logging framework by using a system property at application startup, or at runtime with an explicit API call.

The system property name is `com.genesyslab.platform.commons.log.loggerFactory`. It may contain the FQCN of your particular implementation of the Platform SDK common loggers factory, or the alias name of a built-in implementation (such as "log4j2", "slf4j", etc). If you specify one of the alias names, this value is also used by the `LmsLoggerFactory` initialization logic, so that LMS events from `LmsEventLogger` will also be directed to the same logging framework.

In this case, the jvm option might look like:

```
-Dcom.genesyslab.platform.commons.log.loggerFactory=log4j2
```

This property is handled with the Platform SDK customization options, so you can also enable it by adding a PlatformSDK.xml file with the following contents to your application classpath:

```
<?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">log4j2</entry>
</properties>
```

Changing this property after initialization of the `LoggerFactory` has no effect. So if you need to enable logging or change the target framework at runtime, this should be done with an explicit API call. In this case the Platform SDK commons loggers and LMS events loggers need to be reconfigured separately:

```
// PSDK Commons loggers (re-)configuration:
Log.setLoggerFactory(Log.LOG_FACTORY_LOG4J2);

// AppTemplate LMS Events loggers:
LmsLoggerFactory.setLoggerFactoryImpl(Log.LOG_FACTORY_LOG4J2, lmsMessageConveyor);
```

> **Important**
>
> Be aware that re-initializing the LMS Loggers factory requires a reference to the actual `LmsMessageConveyor`. It is also possible to use "null", in which case the factory initialization method will try to reuse reference from the "current"

> `LmsLoggerFactory` or will create a default conveyor instance with support of the `CommonLmsEnum` events.

These calls are enough to reconfigure loggers which were created earlier; there is no need to recreate them.

Configuring Logging

Platform SDK does not write logs itself (that is, it's not about the legacy Logger Component). Instead Platform SDK is just able to send logs to the specified logging framework where they will be handled. Configuration of logging parameters such as log file names, log levels, and more may be done within that framework.

The Application Template application block contains parsing logic for Genesys Configuration Manager Application logging options properties, and Log4j2 configuration structures to allow automatic framework configuration. This helps applications to automatically start logging to the recommended Log4j2 framework (as discussed below).

It is also possible to create user defined configurators for some other framework. In this case your application may use the Application Configuration Manager to retrieve application configuration details from Configuration Server in the form of POJO structures, and apply those details to its custom logging framework.

Configuring Logging with Log4j2

Log4j2 is the recommended logging framework. AppTemplate provides several options for the logging framework configuration.

Using the Application Configuration Manager allows automatic Log4j2 configuration and reconfiguration in accordance to the Genesys Configuration Manager application logging options.

Beside the common Genesys logging options, AppTemplate also supports a custom "log4j2-config-profile" option, which allows you to create combined logging configuration as a merge of user defined loggers/appenders with ones created by the Configuration Manager application options. This is useful in cases where your application consists of several sub-systems, and is required to split logs from those sub-systems to different log files.

For example, in a Web Application it may be reasonable to separately write logs from Tomcat, Cassandra, Platform SDK, and the application itself. In this scenario, one straightforward way of logging configuration may be following:

1. Create a "startup" log4j2.xml configuration, which will be used before the application has retrieved information from configuration server. This configuration might contain declarations of the application-specific loggers and appenders. Appenders may be either "startup" or "permanent". The first ones have names starting with "PSDKAppTpl-".
2. When your application starts with enabled Log4j2 usage, it picks up and uses the startup

configuration "as is". So, we have the application startup logs.

3. When your application has received the Genesys Configuration Manager Application options, it creates and applies `PsdkLog4j2Configuration` where `log4j2-config-profile = log4j2.xml`. It takes the log4j2 configuration as a base and replaces its startup appender(s) with new ones from the Configuration Manager Application logging options. This allows you to configure the application logging parameters in accordance to Genesys Configuration Manager Application logging options, and gives you the ability to handle external (for example, Tomcat or Cassandra) logs separately if desired.

.NET

The additional features described in this article are not applicable for Platform SDK for .NET.