Agent Interaction SDK Java Developer Guide

Additional Details

12/18/2025

Contents

Additional Details

This chapter describes how to manage several categories of data that the AIL library provides.

Attached Data

User data, or attached data, can be any data attached to an interaction. For example, an IVR transaction may generate attached data associated with a phone call.
Attached data has the following characteristics:

- It is one or more key-value pairs.
- It is available for the whole life of an interaction—it exists in the interaction from its creation till its end.
- The API has features for managing attached data.
- Attached data can be saved in the history as part of the call, once the call is released and marked as done.

Because an attached data is a writable key-value map, it can be any data useful to your application's design. However, it can also include the following specific attached data:

- Interaction attribute values.
- Custom attached data's values.

The Configuration Layer defines keys and information for this attached data, available through the `InteractionManager` interface, as detailed in the following subsections.

InteractionManager

The `InteractionManager` interface gives access to metadata information describing interactions' attached data. The Configuration Layer defines this information in the `Business Attributes` section.

Custom Properties

The `Interaction Custom Properties` in the Configuration Layer correspond to the `CustomAttachedData` objects that your application can retrieve using the `InteractionManager.getAllCustomAttachedData()` method.
The `CustomAttachedData` class describes a single custom property. This class includes methods to get the corresponding name, display name, and description of a custom property. It also provides the predefined values for the custom attached data (if any).
Call the `CustomAttachedData.getName()` method to get the name of a custom property and use it as a key to access or modify the corresponding value in an interaction's attached data map.

Interaction Attributes

The`Interaction Values` in the Configuration Layer correspond to the `InteractionAttributeMetaData` objects that your application can retrieve using the `InteractionManager.getAllInteractionAttributeMetaData()` method.

The `InteractionAttributeMetaData` class describes an interaction attribute. This class includes methods to get, for example, the corresponding name, display name, and description of a custom property. It also provides the predefined values for the attribute (if any).

Call the `InteractionAttributeMetaData.getName()` method to get the name of an interaction attribute and use it as a key to access or modify the corresponding value in an interaction's attached data map.

> **Important**
>
> These attributes can be used to retrieve interactions from a contact history. See Contact History.

Handling

The API provides you with a set of methods dedicated to attached data in the `AbstractInteraction` superinterface. All `Interaction` interfaces extend the `AbstractInteraction` superinterface. The following code snippet shows an example of how to create or set new values for the user data attached to the `InteractionVoice` object:

```
// creation of an Interaction
InteractionVoice voice = (InteractionVoice)
mAgent1.createInteraction(MediaType.VOICE, null,Queue);
voice.makeCall( DN2,null,InteractionVoice.MakeCallType.REGULAR,null,null,null);
//...
// Setting or adding new values
voice.setAttachedData("1", "one");
voice.setAttachedData("two", new Integer(2));
//Saving changes
voice.saveAttachedData();
```

If your application calls a `setAttachedData(String` or `Object)` method to modify some attached data, save the attached data by immediately calling the `AbstractInteraction.saveAttachedData()` method to commit all modifications on key-value pairs in the database and the T-Server.

> **Important**
>
> If your application uses the `setAttachedData(Map)` method passing in all the key-value pairs in the `Map` argument, there is no need to save attached data. The changes are committed when calling the method.

You can also create and fill a Map, then pass its reference in as a parameter of a call method. This is illustrated in the following code snippet in a `makeCall()`:

```
HashMap userData = new HashMap();
userData.put("3", "Three");
voice.makeCall( DN2,
                null,
                InteractionVoice.MakeCallType.REGULAR,
                userData,
                null,
                null);
```

Important

Your program can be notified of an attached data change when an Event occurs. Use the Extension Map and the `ATTACHED_DATA_CHANGED` key to retrieve the data of interest. For details, see the Event-AIL Data section immediately below.

Event-AIL Data

Within `InteractionEvent` events, the library propagates additional AIL information called Extensions. They are different from TEvent Extensions. AIL Extensions can be retrieved through dedicated methods.

The `InteractionEvent.getExtensions()` method returns extended information about the event in a Map. Any keys present in this Map are defined in an `InteractionEvent.Extension` enumeration.

The following code snippet shows how to access an Extension in a transfer context. It implements an Agent handler, which takes into account the possibility of a transferred call ringing and manages the corresponding extension.

```
//Implementation of the Agent.HandleInteractionEvent() method
public void handleInteractionEvent(InteractionEvent _ie) {
        //Retrieval of the map containing the AIL Extensions
        Map extensions = ie.getExtensions());
        //Current status
        Interaction.Status eventStatus=interaction.getStatus();
        switch(eventStatus.toInt()) {
                //...
                // The interaction is ringing case
                Interaction.Status.RINGING_: {
                        // Retrieval of the possible transfer
                        String transferReason = (String) extensions.get(
InteractionEvent.Extension.RINGING_TRANSFER_REASON);
                        // Test if there is a transfer reason
                        if(transferReason!= null){
                                // Display of the corresponding reason
                                System.out.println("Transfer reason" +transferReason);
                        }
                }
                break;
                //...
        }
}
```

See the Javadoc API Reference for details on `InteractionEvent.Extension` keys.

Log Management

The Interaction SDK's log management is based on the `org.apache.log4j` package. The following sections first describe the default log level provided, and then describe the log system in the library.

Default AIL logs

This section discusses the default log level provided. It introduces the `log4j` package and the default log features in the AIL library.

log4j

log4j is an open-source tool designed to help write log statements to a variety of output targets. The AIL library uses the `org.apache.log4j` package to write traces to log files and to the console.

Log4J instantiation and bootstrapping are done internally by the library. You do not have to write code to perform these tasks.

The AIL library uses the main components of this package and follows Apache recommendations. The log4j version number is available in the `log4j.jar` file delivered with the Interaction SDK.

Warning

Genesys does not provide any technical support for the `org.apache.log4j` package.

AilLoader

By default, the Interaction SDK provides you with console and file traces. You can access these default logs with the `AilLoader` interface.
The `AilLoader` class enables you to:

- Disable the logs with the:
    - `AilLoader.noTrace()` method for the console.
    - `AilLoader.noLogFile()` for the log file.
- Set a debug level for the traces.
- Set your log file location.

Please refer to the AIL Javadoc API Reference for more details.

Warning

If the debug level for the traces is defined in the Configuration Layer, the library core will take this level into account upon connection to the Configuration Layer.

Adding Logs

You can add logging to your application with or without using the `log4j` package. AIL does not require you to use `log4j` for your own system trace. If you choose to use `log4j`, you can follow the recommendations in this section. For further information, refer to Jakarta documentation at:`http://jakarta.apache.org/log4j/docs/documentation.html`.
The following subsections discuss how you can use the `log4j` package to:

- Mix your own traces with the library traces.
- Generate your own traces separated from AIL logs.

Mixed Traces

You can choose to use `log4j` to add your own traces to the log, in order to mix them with AIL-generated traces. For example, you can use the Root Logger object of the `org.apache.log4j` package. The following code snippet uses the Root Logger that has already been internally instantiated by the library:

```
// Retrieving the root Logger
LoggerRepository mLoggerRepository = LogManager.getLoggerRepository();
Logger mRoot = mLoggerRepository.getRootLogger();
// Defining a layout
PatternLayout layout = new PatternLayout("%d{dd MM HH :mm:ss:SSS} [%20.20t] %-5.5p %20.20c
%m%n");
// Creating a FileAppender object to append the logs
// events occurring.
FileAppender mFile = new FileAppender(layout, "./myFile.log");
mFile.setThreshold(Level.DEBUG);
// Adding your FileAppender to the Root
mRoot.addAppender(mFile);
// Adding a message of level debug:
mRoot.debug("**** My debug message! ****");
```

Separated Traces

You can also use `log4j` to create separated logs. You just have to create your own `Logger` object, as shown in the following code snippet:

```
// Creating the Logger
Logger mLogger = Logger.getLogger("myFile.Log");
PatternLayout layout = new PatternLayout("%d{dd MM HH :mm:ss:SSS} [%20.20t] %-5.5p %20.20c
%m%n");
// Creating a FileAppender object to append the logs
// events occurring.
FileAppender mFile = new FileAppender(layout, "./myFile.log");
```

```
mFile.setThreshold(Level.DEBUG);
// Adding the FileAppender to the Logger
mLogger.addAppender(mFile);
// Adding a message of level debug:
mLogger.debug("**** My debug message! ****");
```

Warning

All the previous code snippets are for illustration purposes only. Code examples are not tested and not supported by Genesys.